# COORDINATED SYSTEM LEVEL RESOURCE MANAGEMENT FOR HETEROGENEOUS MANY-CORE PLATFORMS

A Thesis
Presented to
The Academic Faculty

by

Vishakha Gupta

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
December 2011

# COORDINATED SYSTEM LEVEL RESOURCE MANAGEMENT FOR HETEROGENEOUS MANY-CORE PLATFORMS

Approved by:

Professor Karsten Schwan,
Committee Chair
School of Computer Science
*Georgia Institute of Technology*

Professor Karsten Schwan, Advisor
School of Computer Science
*Georgia Institute of Technology*

Professor Calton Pu
School of Computer Science
*Georgia Institute of Technology*

Professor Sudhakar Yalamanchili
School of Computer Science and
Electrical and Computer Engineering
*Georgia Institute of Technology*

Rob Knauerhase
Research Scientist
*Intel Labs, Hillsboro, OR*

Doctor Jeffrey Vetter
School of Computational Science and
Engineering and Oak Ridge National
Laboratory
*Georgia Institute of Technology*

Date Approved: 22 August 2011

*To my parents,*

*sister,*

*and my rock, Romain.*

# ACKNOWLEDGEMENTS

I would like to thank all people who have helped and inspired me during my doctoral study.

I was told before starting a doctoral degree that my entire experience and learning will depend on the one person, my advisor. And I would have to say, I completely agree. In the past five years, I have learnt so much, I have faced challenges I thought did not exist and I have had times where I could have lost faith in my abilities. But the one important person, my advisor, Prof. Karsten Schwan was always there, to teach me, to show me the right direction and make sure I never lost confidence in my ability to do useful research. There were times when the paper rejections got so unnerving that I would question the purpose of the overall research. He made sure I realized the importance by participating in the paper submissions with equal zeal each time and presenting the work to our collaborators from the industry to show me how my research mattered. He gave me enough choices and independence to define my own problems but made sure I was not barking up the wrong tree. I owe my knowledge, confidence and network of exceptional contacts to Prof. Schwan and I cannot express enough gratitude towards him for making this duration of my degree, the most intellectually stimulating and enjoyable time in my career so far.

Prof. Sudhakar Yalamanchili, Prof. Calton Pu, Dr. Jeffrey Vetter, and Rob Knauerhase deserve a special thanks as my thesis committee members and advisors. In particular, I would like to thank Rob for flying down to Atlanta from Oregon to attend my defense and for his insightful comments throughout my internship at Intel Labs and afterwards too. He was the reason I managed to get access to proprietary Intel hardware without which, part of my research would have lacked the appeal it has now. These people have helped me shape an approach that will hopefully lead to interesting research for future students and inspire solutions for future systems. I would also like to thank Ada Gavrilovska who has

been a good colleague and an excellent friend throughout my PhD. I would really miss our conversations about various things. She is someone I could rely on being awake at 2am in the night if we were working on a paper. Susie McClain, our group admin, has also been extremely helpful and nice. I would like to thank her for all her support in taking care of our travels, day to day lab needs and so on.

I was fortunate to get a chance to intern with really bright minds at some of the most famous research labs in this country. My first encounter with an extremely smart and thorough researcher was at IBM T.J. Watson Research Center. I learnt a great deal from Jimi Xenidis in the summer of 2007. I am very thankful to him for getting me started on the deeper ideas of processor architectures and system virtualization. I would also like to thank my manager Dilma Da Silva, who is still a mentor to me.

Summer 2008 was another lesson during the PhD at HP Labs. Niraj Tolia and Vanish Talwar prepared me for the tough questions that can be posed before a researcher when presenting any new idea however brilliant it might seem to the one proposing the idea. It might seem like an easy task to defend an idea that seems to have great potential but I was surprised by the range of questions people ask, very often due to their years and years of experience. Their encouragement and guidance was invaluable in learning how to anticipate these concerns and calmly respond with satisfactory answers. Niraj was also quite helpful in giving me general advice for a successful PhD life. I thank them and my manager Parthasarathy Ranganathan for making sure our work got published at the top venue for Systems research.

I worked on the favorite part of this thesis starting Summer of 2009 and continuing into Summer 2010 at Intel Labs. Rob Knauerhase and Paul Brett were very encouraging when I presented my ideas for a different kind of scheduler. I would like to express my heartfelt gratitude to Paul, without whom it would have taken me months to develop insightful experiments and learn the complexities of Intel architecture. I would also like to thank my managers during those two summers, Scott Hahn and Jeffrey Jackson. I met some very

smart and extremely cordial people during these two summers and I am looking forward to interacting with them when I start working at Intel.

My deepest gratitude goes to my family for their unflagging love and support throughout my life. I am indebted to my father, Dr. Suresh Gupta and mother Vijaya Gupta, for their encouragement and support since childhood. I would not have reached this point in my life if it were not for my dad's faith in me, even in a conservative world where people hardly believed in such higher education for girls. He never discriminated and always helped me economically, morally and emotionally in fighting the various challenges that came my way. He made sure I never faced the hardships that he had to, while growing up. My father's example has taught me to fight hard, create my own opportunities, and to never give up. And I can obviously not forget to thank the mom I so dearly love and miss. She has shaped my personality in ways I cannot thank her enough for. All through her life, she has been an epitome of optimism, cheerfulness and fighting spirit. She made sure I remained humble regardless of any achievements. She told me to care for people and never lose that smile on my face even in adversity. She taught me to respect people and fight injustice. She is the reason I can wade through puddles of tears and come out seeking the sunshine. I am proud to be their daughter and I love them very much. I would of course be committing a crime if I did not thank my dear little sister Sonam Gupta. Since her childhood, she has been more excited about my career than anyone else. She would make sure I never exceeded my stipulated nap durations, she would pack my pencil box before my board exams and she would always speak to me as if I could never fail. Her faith and encouragement has come a long way with me. I call her funny names and tease her despite being older. And yet she has never lost respect and has always made me feel a responsible and capable person. She is the greatest sister one could ever ask for.

This acknowledgement would be incomplete without a big thank you to my husband, Romain Cledat. I met him in my first semester at Georgia Tech and somehow I felt like he could be a friend I would cherish. Well, over time he proved to be someone I could share

my joys and sorrows with, someone I could carry out technical discussions with or someone I could just talk about life with. I am sure I couldn't have had the most insightful graphs in my thesis without his help. I learnt so much about scripting and performing experiments from him and these are skills that will assist me throughout my career. Romain, thank you for your immeasurable faith, love and respect. Not only did you provide me with your incessant support but also with insight and much needed guidance. You are my idea of bliss and I am privileged to be your wife and best friend.

Thomas Jefferson said "But friendship is precious, not only in the shade, but in the sunshine of life, and thanks to a benevolent arrangement the greater part of life is sunshine." And I have been blessed to have a cozy group of friends, at different stages of my life, to bask in the sunshine with or to get through the darkness with. These friends have been responsible in helping me define myself with respect to the world we live in. Words are not enough to thank my friends from childhood, undergrad, masters and PhD. I have known Roopal Sarda since I was in seventh grade and she has always been someone I can talk to about anything that bothers me or anything that pleases me. Distance has not diminished our trust and friendship. She has helped me in so many different ways all through my graduate life. I cannot thank her enough for just being there. Abhijit Chavan and Radhika Chavan have always been there to make me laugh and realize that no matter where life takes us, those who truly care, remain the same. I would also like to thank Rahul Iyer and Aadesh Desai for their supportiveness when I was making a choice between PhD vs. job. They have been a part of my happiness and life.

When the lab and the building you work in has people who can make technical contributions as well as make you feel alive in a fun social setting, it makes five years seem like a short duration. I have Mukil Kesavan, Adit Ranadive, Dulloor Rao, Danesh Irani, Bhuvan Bamba and Himanshu Raj to thank for that. Vijay Balasubramanian was my first friend at Georgia Tech and he is very special because I have known him through these five years and he was the one who made sure I did not go to eat lunches alone. He has been

an excellent friend ever since and he introduced me to more people who made a difference like Raghavendra Vijaywargiya and Rakshita Agarwal. I will never forget my most interesting coffee breaks at Octane with Tushar Kumar. We had discussions on politics, control systems, math formulas and paper reviews. He has been of tremendous help and a very sweet friend all throughout. I would also like to thank some of my other friends like Ashwin Kolhe, Amit Tambe, Sanjay Kumar, Min Lee and Vishal Gupta, who have been of great help from time to time. Ashwin and Amit have been very dear to me since we started together at Georgia Tech.

Get a bad roommate and student life is hell. Get a good roommate and it's all fun. And I can say with great pleasure that I have been blessed with the best roommates who are still among my closest friends. I have learnt so much from Madhavi Wagh. She always has the most sound advice I can get on life issues. Her talent in graphic design still eludes me but she has taught me how to appreciate art. She was the best roommate and I am extremely thankful to her for all the food she used to leave out for me when I worked late during my first two years. And I cannot thank Sucharita Otta enough for the hours of gossip and light hearted conversations that brightened my mood even after a long hard day. Priyanka Tembey was my labmate, collaborator and my roommate. She would stay late in the lab during those paper deadline nights to support me and calm me down when I shuddered close to the deadlines. She would read my papers when I would start phasing out due to fatigue. She would come jumping around to celebrate with me when I made a breakthrough. I have shared precious moments with all my roommates and I thank them for being the greatest roommates one could ever ask for.

Last but not least, I thank God for helping me get through all the tests in the past five years. I have always imagined God to be a powerful force in the world that keeps me humble and provides an inner strength because I look up to him in troubled times and thank him when I achieve the desired results. Realizing the wealth of knowledge, the warmth in relationships and appreciating the beauty of the world around me has always

kept me excited in life and wanting for more such experiences.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# SUMMARY

Heterogeneous multi-cores—platforms comprised of both general purpose and accelerator cores—are becoming increasingly common. Contemporary processor designs have also shown a rapid increase in the number of cores per chip. Recently, this trend has begun to include functional and performance asymmetries to balance power vs. performance requirements. Effective use of such heterogeneous resources, however, requires suitable system abstractions and methods to manage them for the dynamic and diverse workloads imposed by applications. This poses new challenges in platform resource management, which are further exacerbated by the need for runtime power budgeting and by the increased dynamics in workload behavior observed in consolidated data-center and cloud-computing systems.

Hence, the first problem addressed by this dissertation, for both heterogeneous and asymmetric future chip architectures, is *how to convert this heterogeneous pool of resources to a manageable many-core platform*. The hardware heterogeneities considered arise from on-chip asymmetries, e.g., NUMA effects of memory, varying frequencies of cores on the same die etc., or are due to the presence of accelerators like GPUs, which implies that platform cores differ widely in their compute speeds, memories, and access speeds. Contemporary solutions for dealing with heterogeneity wrap such hardware inside proprietary drivers and runtime systems to abstract complexities from application and systems developers. Our research has shown that hiding processing units like GPUs behind a driver precludes their efficient utilization and sharing. Thus, the first contribution of this research extends virtualization technologies to make it possible to treat accelerators as first class schedulable resources, just like the platform's general purpose processing cores. We

also extend this argument to heterogeneities encountered on-chip. In short, we propose and evaluate methods to make such resources visible to systems software and also make a case for collaboration between different layers of the software stack in order to enable high application performance and effective system utilization.

Once manageability is established, this then leads us to solve the problem of *how to schedule resources* and of *how to do so efficiently*, in order to meet application needs and system goals like high performance for a wide range of workloads. Challenges addressed by the second step taken in this dissertation arise from the differing requirements of applications or from different goals of platform resource managers. For example, these could include (1) attaining maximum performance with existing system resources, or (2) meeting some application-specific metric or service level objective (SLO). In this context, applications can span the spectrum of serial to parallel, client to high performance, those written for homogeneous hardware or those adapted for heterogeneous hardware like CUDA-based codes that can run on x86 cores as well as NVIDIA GPUs. Applications can also differ due to the programming models they use or the execution models on which they are based. The second contribution of our research is the development of a resource management framework and representative resource management methods with which application needs or platform goals can be attained.

The implementation of our approach leverages the increasing presence of virtualization in data center, HPC, and even desktop environments. Therefore, to achieve the goals outlined above, we develop hypervisor-level support for **hybrid virtual machines (HyVMs)**. Virtual machines with different personalities are called 'Hybrid Virtual Machines'. *A VM personality symbolizes its execution context with attributes that match a particular type of processor(s)/core(s) in the platform.* That is, a VM's personality captures the differences in the execution of the VM across heterogeneous cores. In the simple case a VM can just execute with general purpose virtual CPUs (VCPUs), thereby defining a VM with a single 'general personality'. A more interesting example is when the VM runs accelerator code:

it uses an *accelerated or compute personality*, expressed by activation of one or multiple of its accelerated virtual cores. In this case, the accelerated VCPU forming its 'compute personality' co-exists with its general-purpose VCPUs that define its 'general personality'. VMs may have multiple personalities or their personalities may change (with performance consequences) as they are migrated and consolidated across cores of various types and capabilities. In this dissertation, we present our *Pegasus* architecture for demonstrating the notion of VM personalities on a heterogeneous platform comprised of x86 and GPU cores. We implement and evaluate a second complementary system for asymmetries on chip called *Montage*. The techniques, resource management methods and lessons learnt from both the prototypes will be applicable to future hybrid systems that will combine the power of on-chip asymmetric multi-cores with off-chip accelerators or platforms with a wider range of heterogeneity in processor and memory resources on-chip.

# CHAPTER I

# INTRODUCTION

Increasingly important constraints on power consumption coupled with the highly diverse performance expectations and execution characteristics of modern applications have resulted in a spectrum of heterogeneity for modern processor configurations, as depicted in Figure 1. Processor architectures have evolved from homogeneous systems comprised of replicas of identical cores to those with shared or single ISA heterogeneity [53, 64] on the same die. Contributing to this trend is the proven utility, in terms of performance and power consumption, of specialized processors like those used for accelerating computations, network processing, or cryptographic tasks leading to an integration of heterogeneous cores like network and graphics on chip with general purpose or shared ISA cores [4, 40, 43]. The evolution of hardware depicted in Figure 1 has been inspired and well supported by the domains of (1) high performance computing (HPC) with concerns of improved computational capabilities at reduced power costs, (2) mobile platforms with limited battery capacities, (3) cloud computing or utility data centers that aim to provide services to a wide spectrum of workloads and wish to maximize power to performance ratios, and (4) high end client workstations like those used for gaming or graphics development.

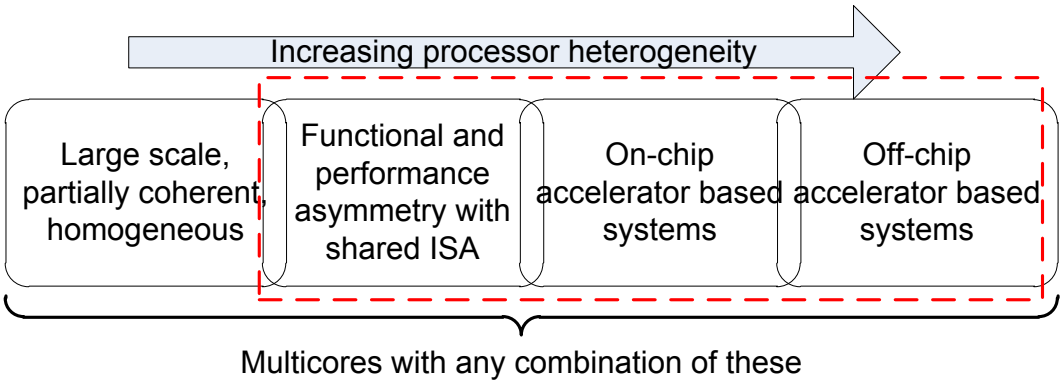
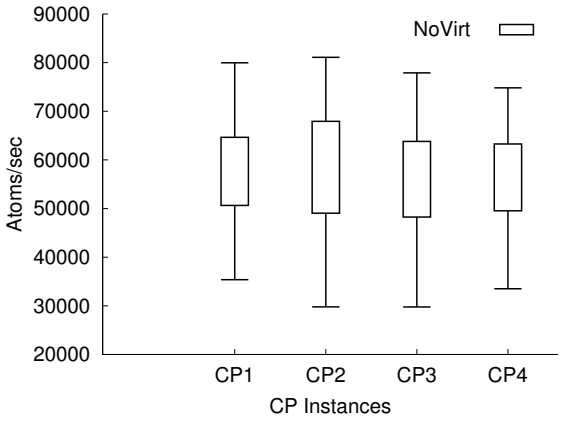



**Figure 1:** Evolving spectrum of processor heterogeneity

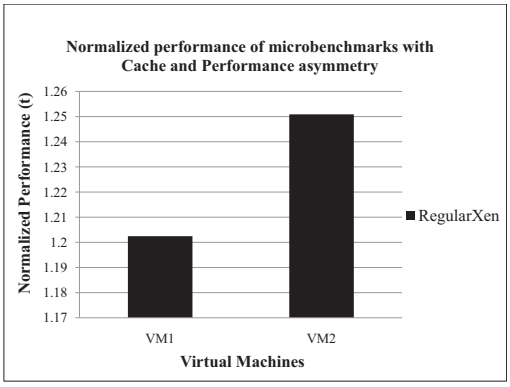
However, this trend presents disruptive challenges to systems software, including how to service this range of heterogeneity without requiring repeated changes to operating systems or applications, and how to deal with hardware evolution, in ways that attain high levels of utilization and application performance. In particular, this trend represents a significant departure for mainstream virtualization technologies, founded on a fixed ISA, that host multiple virtual machines. The primary issue faced by the systems software running on such heterogeneous computing platforms is - in what manner and to what extent to expose to systems and applications, the diverse nature of underlying hardware. Current solutions range from approaches in which systems expose accelerator 'devices' with their own tool chains – henceforth termed *heterogeneous architectures* – to those that simply accelerate certain instructions added to, or already part of, the instruction set architecture (ISA) – termed *asymmetric architectures*. While the presence of such heterogeneous cores, whether on-chip or off-chip (connected via high speed interconnects), can provide a resource pool suitable for a wide range of applications, the absence of comprehensive runtime methods for resource management can often lead to under-utilized or inefficient use of this rich set of processing units.

For example, typical accelerators have a sophisticated and often proprietary device driver layer, with an optional runtime. While these efficiently implement the computational and data interactions between accelerator and host cores [77], they lack support for efficient resource sharing, as shown by large variations in measurements of the throughput achieved, with applications executed directly over CUDA runtime and the NVIDIA driver in Figure 2.a) and sharing two GPUs between four applications. Figure 2.b) demonstrates this when scheduling VMs to run on a shared ISA asymmetric machine using regular Xen [8] VCPU (virtual CPUs exposed to guest virtual machines) scheduling, which is unaware of platform asymmetry. Both examples demonstrate a need for explicit resource management with platform-aware resource scheduling policies.

Hence, sustaining software productivity gains through the evolution to asymmetric and




a) Variation in Coulomb Potential [CP] benchmark [90] due to sharing




b) Far from optimal performance on an asymmetric platform with processor speed and socket cache asymmetry [with optimal = 1]

**Figure 2:** Need for manageability and better resource scheduling in heterogeneous multi-core platforms

heterogeneous architectures requires innovative approaches to virtualize and manage resources in such environments. To better manage the resources of evolving asymmetric and heterogeneous hardware architectures, our research proposes hypervisor-level support for 'hybrid' virtual machines (HyVM), where differences in the execution of a VM across heterogeneous cores are captured by the notion of VM *personalities*. Specifically, *a personality is a quantifiable set of attributes that determines or influences the way an executable entity is matched with the hardware's processing resource*. For a HyVM, this means, therefore, that its executable units – its virtual CPUs (VCPUs) – have identifiable personalities, and it is these VCPUs with personalities that are mapped to (i.e., matched with) the heterogeneous physical CPUs (PCPUs) present on underlying hardware. Virtual machines composed of different or multiple personalities are called *Hybrid Virtual Machines*.

As an illustrative example, a program written with CUDA or OpenCL, for instance, has a GPU personality, and it must run on processors supporting the ISA generated by compilers (e.g., the binary translation of PTX codes). In contrast, a program compiled into x86 binaries must run on cores supporting the x86 ISA. Moreover, when running applications

on current GPGPU-based hardware, resource management must actually schedule two different personalities: (1) that of the host program issuing GPU calls and (2) the execution of code on GPUs. Stated more explicitly, for GPGPU-based systems in which both types of processors – general purpose and accelerator cores – are placed into a single schedulable pool of processing resources, the hypervisor must schedule VCPUs with two different personalities: a general purpose and an accelerator personality. More generally and extending the personality concept to other types of heterogeneous hardware, there may also be more subtle – and more fluid – differences between VM personalities, an example being a personality characterized by its frequent use of the SSE instruction, thus requiring SSE instruction support from the underlying hardware, in contrast to one that only infrequently or never uses it. Further, from this example, it should also be clear that VM personalities can change over time, since each VM can run different and multiple applications and since even a single application may have different execution phases or changing runtime behavior. In all such cases, it is the hypervisor that performs the runtime matching of personalities to processing cores. This approach can efficiently exploit the resource diversity on heterogeneous hardware because:

- it departs from current device-based models for using accelerators like NVIDIA's GPUs, by creating an execution model in which all platform processing components become 'first class schedulable entities', resulting in a manageable pool of resources controlled by the hypervisor;

- it improves the fungibility of these resources, by making it possible, for instance, to run computations on multiple 'types' of target cores, thereby providing additional levels of flexibility to hardware designers for making suitable platform-level power/performance tradeoffs; and

- it permits creation of diverse scheduling and resource management strategies to efficiently use the pool's heterogeneous resources for different platform and application

workloads or requirements.

The implementation of our proposed approach leverages the increasing presence of virtualization in data center, HPC, and even desktop environments, or more specifically, it exploits the presence on modern platforms of specialized hardware that supports virtualization. As a result, the lowest layer of software present on the heterogeneous platforms considered in this thesis is the hypervisor and its management (or driver) domain (if present). HyVMs represent a powerful system management extension to state-of-the-practice virtualization technologies. We demonstrate the effectiveness of the HyVM abstraction through our example implementations:

(1) **Pegasus**: implemented on an x86-GPGPU accelerator-based system. The Pegasus hypervisor extensions make accelerators into first class schedulable entities and support scheduling methods that enable efficient use of both the general purpose and accelerator cores of heterogeneous hardware platforms. Specifically, for platforms comprised of x86 CPUs connected to NVIDIA GPUs, these extensions can be used to manage all of the platform's processing resources, to address the broad range of needs of GPGPU (general purpose computation on graphics processing units) applications, including the high throughput requirements of compute intensive web applications and the low latency requirements of computational finance [69] or similarly computationally intensive high performance codes. For high throughput, platform resources can be shared across many applications and/or clients. For low latency, resource management with such sharing also considers individual application requirements, including those of the inter-dependent pipeline-based codes employed for the financial and image processing applications. The manageability methods and resource management schemes implemented in Pegasus are described in Chapter 4.

(2) **Montage**: implemented on shared-ISA asymmetric cores (Chapter 2). The Montage system described in Chapter 5, addresses the challenges and opportunities presented by *asymmetric* multicore platforms, i.e., by platforms with shared or single ISA cores that

differ in their performance or functional properties. For such processors, Montage implements an approach, framework, and methods for creating asymmetry-aware hypervisor. With Montage, arbitrary guest virtual machines (VMs) and their applications can efficiently run on asymmetric hardware. Furthermore and as a consequence of its hypervisor-level realization, as platforms continue to evolve, Montage can shield guest operating systems and applications from recurrent porting for each new generation of asymmetric hardware deployed in the cloud or data center infrastructure. This is done by enriching the hypervisor with methods that manage VM execution in ways that are cognizant of workload characteristics in order to best leverage asymmetric platform resources. In particular, Montage offers the novel method of *kinship-based scheduling*, which generalizes prior affinity-based scheduling to also take into account platform asymmetries and workload phases.

We have developed two separate systems for demonstrating our techniques because of the evolving nature of hardware research and its availability. HyVMs are an abstraction suitable for future hybrid systems. Contemporary hardware is still not at a stage where all of the techniques could be tested simultaneously without large porting efforts. But Pegasus and Montage serve as vehicles to exemplify VM personalities and to emphasize the following technical contributions made in this thesis:

- *obtaining manageability:* by creating a common pool of heterogeneous resources explicitly controlled and monitored by the hypervisor;
- *improving fungibility:* by giving VMs multiple personalities and then adding hypervisor-level software to flexibly map them to, and efficiently run them on, heterogeneous hardware;
- *range of scheduling policies:* for efficient utilization of this managed pool of resources while meeting different system goals; and
- *cross layer communication:* for sharing relevant information across the systems stack, if possible (useful but not required), to improve performance.

The first two contributions are key for the creation of Hybrid Virtual Machines (HyVMs) as the basis for virtualization of heterogeneous cores. The latter provide diverse resource management strategies needed to sustain the evolution of heterogeneous architectures and systems and the applications that use them. Finally, the HyVM approach also offers APIs with which applications and guest operating systems can express their desires and reactions concerning such hypervisor-level resource management as discussed in Chapter 6.

## *1.1 Thesis Statement*

During the course of this research and through our evaluation of hybrid virtual machines, we have identified that, for efficient utilization of future heterogeneous platforms, it is important to recognize their ISA and architectural differences at each level of the systems software stack. System mechanisms and abstractions should support explicit management of heterogeneous resources to meet application needs and platform requirements. The resulting heterogeneity-aware runtime methods for managing system resources can substantially improve resource utilization and application performance, including those that coordinate resource management across different system silos, when managing architectures in which accelerators and general purpose processors reside on different chips or in different coherence domains. Finally, additional benefits are derived from runtime sharing of information about hardware state and application or workload across different levels of the software stack.

The remainder of this dissertation is dedicated to quantitatively proving the thesis statement and is organized as follows. Due to the complexity and range of hardware considered in this research, we first present our machine model in Chapter 2. Chapter 2 also lays out the application domains that can benefit from the concepts presented and evaluated in this thesis. Chapter 3 outlines our research contribution through the proposed HyVM architecture for a representative heterogeneous many-core platform. We present the implementation and evaluation of VM personalities on heterogeneous systems in Chapter 4 and

asymmetric systems in Chapter 5. Chapter 5 also describes scheduling methods that can be extended to work with the entire range of heterogeneity targeted by this research. The APIs for cross system communication for better resource management are described in Chapter 6. Related work appears in Chapter 7 followed by conclusions in Chapter 8 and possible future directions in Chapter 9.

# CHAPTER II

# BACKGROUND

The unifying theme throughout this dissertation is its goal to address the performance requirements of diverse applications and effective utilization of available resources for system administrators, despite increasing heterogeneity in computing platforms. Before we delve into the techniques for doing so, it is important to understand the hardware model assumed throughout the thesis and the application domains it caters to.

## *2.1 Machine Model*

Driven primarily by energy constraints, modern processors have become increasingly asymmetric in their hardware features. For example, even when the instruction set architecture (ISA) as the interface between hardware and software remains the same, the micro-architecture implementations of this ISA can range from complex out-of-order cores to simpler compact in-order cores, each in turn potentially operating across a range of voltage-frequency combinations [50, 53, 63]. We refer to such differences as *performance asymmetries*.

Alternatively, more energy efficient implementations of an ISA may be realized via simpler streamlined data/control paths and by eliminating complex operations such as floating point logic and certain complex instructions can be supported with specialized on-chip logic. This produces cores that support a subset or a superset of the original ISA. We call this *functional asymmetry*. Denoted as the shared-ISA model in [65], HyVMs address the general shared-ISA case, in which the ISAs of two cores overlap considerably. Examples are cores that differ from other processors only in that they may omit/add certain instructions, such as those needed for encryption or network acceleration.

HyVMs can also leverage further gains in energy efficiency and throughput achieved

by sacrificing generality and producing core architectures that implement distinct computing models such as vector and single instruction multiple thread (SIMT), and very large instruction word (VLIW) ISAs. These designs produce an order of magnitude or more improvement in joules/op but (1) sacrifice generality, (2) are efficient for specific classes of computation, e.g., data parallel, and (3) reflect distinct computing models and therefore, typically have distinct ISAs. We refer to architectures with cores supporting (multiple) distinct ISAs as *heterogeneous architectures*, e.g., systems with accelerators like NVIDIA GPUs.

## *2.2 Application Domain*

Increasing numbers and sizes of data centers and cloud installations provide access to high performance computing to common users in the form of gaming clouds [79], financial applications [69], high quality media delivery and processing [74], and others. Conversely, increased demands from high end applications is driving IT providers to create new data-center systems that support such applications using GPU-based accelerators and high bandwidth [1] and perhaps, also low-latency data-center networks or machine interconnects [81]. Recent examples include Amazon's GPU cloud [3] and federally funded GPU-based machines [109], as well as data-center clusters and software offered by vendors that specifically address the needs of high performance parallel codes [83, 80, 2]. The power constraints imposed by the increasing scale of deployments of such systems is also responsible for the recent research on asymmetric platforms. This dissertation targets such systems that witness a diverse combination of workloads.

# CHAPTER III

# HYBRID VIRTUAL MACHINES: USING PERSONALITIES TO VIRTUALIZE HETEROGENEOUS ARCHITECTURES

In this chapter, we talk about the general architecture and mechanisms proposed in this thesis for hybrid systems beginning with a formalization of VM personalities. The following chapters will describe instantiations on accelerator-based systems and on-chip asymmetric systems followed by a general representation of scheduling framework that could guide system development in the future, to keep up with the hardware evolution.

## *3.1 The HyVM Approach*

Asymmetric and heterogeneous processors present specific new challenges to future systems. To address these challenges, we develop and evaluate the notions of HyVMs and their *personalities*.

### 3.1.1 VM Personalities

Given the range of heterogeneity in processors (henceforth termed physical CPUs or PCPUs), we introduce some formalisms to more precisely define HyVMs.

Let $D_i$ denote a VM (i.e., a domain) in the system, where $i = 1 \ldots N_D$ and $N_D$ is the total number of VMs in the system. Similarly, $C_j$ where $j = 1 \ldots N_C$ and $V_k$ where $k = 1 \ldots N_V$ denote processing units or cores in the platform and VCPUs (virtual CPUs) belonging to different domains in the system, respectively. $N_C$ and $N_V$ are the total number of PCPUs and VCPUs in the system and $SetC$, $SetV$ denote their entirety. With platform heterogeneity or asymmetry, additional information is needed to describe these physical and virtual cores, and we represent such information with 'tags' denoting their performance

and functional differences. So, $SetT$ for a platform can be defined as a set of different performance and/or functional behaviors exhibited by the PCPUs in the platform as $SetT = \{T|\text{T identifies different execution attributes}\}$. Therefore, $C_j^T$ denotes PCPUs that are tagged with T where $T \in SetT$. For instance, T could be a tag denoting the presence of vector or cryptographic units, or it could denote the core's classification as a big or small general-purpose core (computed from its various attributes of cache size, processor speed, ...), etc. It is also possible for a PCPU tag to be a union of multiple individual tags, depending on their definition in the system. For instance, if tag $T1$ denotes a fast core and tag $T2$ denotes AESNI crypto support, then a fast PCPU supporting AESNI will carry a tag $T_u = T1 \cup T2$.

Given tags, a personality $P$ can be defined as the set $\{T_u|T_u \in SetT\}$, which is the set of all execution attributes or tags $T_u$ that can influence the matching of an executable entity to a physical resource in the platform. Stated more precisely, a *personality* is a union of all tags that best represent the execution attributes shown by a VCPU. For instance, all VCPUs that can execute on an accelerator like a GPGPU are characterized by the 'GPGPU personality', where $P$ in this case is GPGPU and all VCPUs are personified accordingly. $SetP$ denotes the set of personalities exhibited by various VCPUs in the system. We can now define a VM, or a domain, to be represented as $D_i = \{V^p|p \in SetP\}$, or a set of VCPUs with personalities in $SetP$. For homogeneous platforms with general-purpose cores all characterized by a tag $T^{general}$, VMs can execute only with the single personality $P^{general}$ for all VCPUs. If a VM has one VCPU or all of its VCPUs exhibit the same personality $P$, the VM can be said to have a personality $P$, as well. Domain $D_i$ with VCPUs characterized by multiple personalities represents our notion of a HyVM. Finally, personalities are dynamic, whereas the tags assigned to physical CPUs are fairly static in nature with the possible exception of PCPU speed characteristics modifiable by software. Therefore, we distinguish the characterization using personalities and tags. We denote a VCPU $V$ with some personality $P \in SetP$ as $V^P$.

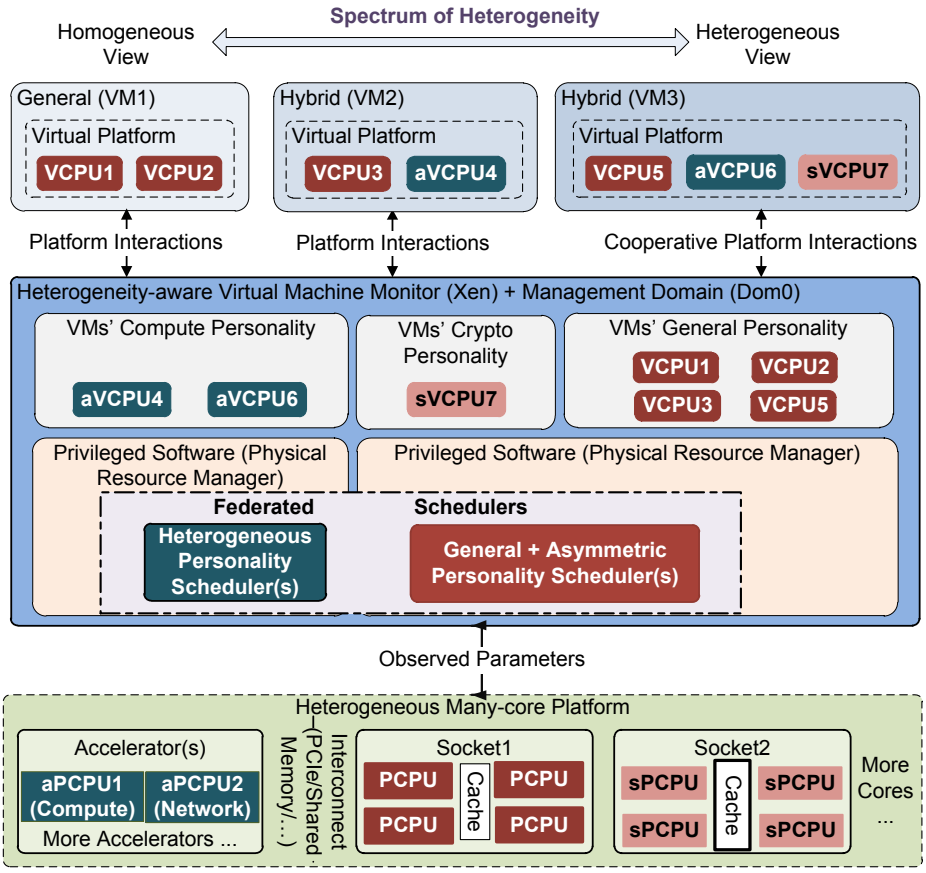



**Figure 3:** HyVM Software Architecture

The hardware shown at the bottom in Figure 3 shows several kinds of PCPUs with $T \in \{compute, network, crypto/shared-ISA, general\}$. For simplicity, they are shown as aPCPU1, aPCPU2, sPCPU, and PCPU, respectively. Similarly, in this simple representation, VMs with their various VCPU personalities are composed of aVCPUs, sVCPUs, and VCPUs. In the straightforward case like VM1 in the figure, all VCPUs can be characterized as general purpose, thereby defining a VM with a single 'general personality'. A more interesting example is VM2, where say $D = \{V^{Compute}, V^{General}\}$. That is, the aVCPU with its 'compute personality' co-exists alongside the VCPU with its 'general personality'. An example of such a guest with two simultaneous personalities is one running applications composed of both graphics kernels running on a GPGPU and x86 host code issuing these kernels via the accelerator API. Such a guest could further be enhanced by a third personality characterizing its sVCPU (VM3 in Figure 3), if say, it executes cryptographic code

running on an SSL-based front end that interfaces to remote users. That code could be accelerated substantially by using built in hardware for encryption, e.g., the AES instruction offered as an extension of the x86 instruction set. The hypervisor could consider this when mapping the guest VM: it could emulate the instruction's execution and hence run it on a normal PCPU, or it could switch the guest from a PCPU not supporting the instruction to one that does, whenever the instruction is used frequently. This is indicated by the sVCPU shown in the 'crypto personality' in Figure 3.

The examples described above show that heterogeneity-aware VMs can explicitly change their personalities, e.g., by activating accelerated virtual cores. However, there are also other and more subtle ways for VMs personalities to change, as described in the next section. A noteworthy point here is that since we are dealing with heterogeneity, we not only have to identify and label the different types of physical and virtual processors, but we may also have to expose those differences to guest operating systems. An instance suitable for this dissertation is the guest being informed about the presence of two GPUs attached to the host machine, thereby enabling it to create multiple aVCPUs for its accelerated personality.

### 3.1.2 HyVM Software Architecture

The concept of VM personalities for dealing with hardware heterogeneity has only recently become viable, because of the increasing prevalence of efficient, hardware-supported virtualization mechanisms on platforms ranging from data-center servers, high-performance (HPC) systems, desktops, to handhelds. Technically, this means that the HyVM approach assumes the lowest layer of software present on heterogeneous platforms to be the hypervisor, with Figure 3 depicting a system based on the Xen hypervisor and its management domain – Dom0. To support the HyVM concept, the hypervisor is extended with the ability to differentiate the VCPU personalities of the VMs it hosts and the tags associated with platform resources, i.e., the sets $SetP$ and $SetT$, for the virtual and physical cores it manages. At the same time, the hypervisor retains considerable flexibility for possible

$VCPU_k^P$-$PCPU_j^T$ mappings, realized through the personality management framework and methods described below.

#### *3.1.2.1 Personality Management*

We have developed two separate systems, for both heterogeneous and asymmetric platforms, in order to demonstrate the generality of the personality concepts and techniques for personality management and scheduling, in the face of the evolving nature of hardware platforms and their availability.

**Heterogeneous personality:** hiding processing units like GPUs behind a driver can inhibit their efficient utilization and sharing, as shown in our system called Pegasus described in Chapter 4. To address this, the HyVM approach first turns accelerators into fully schedulable entities, which can then be managed by the hypervisor-level management policies and actions. This is done by the hypervisor detecting, intercepting, and controlling accesses to the accelerator, either directly, or by relying on *management extensions* that interact with specialized accelerator-level runtimes. For instance, again consider VM2 from Figure 3 and its two personalities. The hypervisor already controls the mapping of the VM's VCPUs, and it gains control over the VM's accelerator use by intercepting its CUDA calls [77] and in response, creating the aVCPUs it can control. The topmost mechanism in Figure 4 shows the direct execution of accelerator code from the VM's virtual platform on the accelerator in the physical platform, although the access to the physical accelerator is controlled by the hypervisor.

Personality scheduling in such systems can then be enhanced by coordinating the scheduling of the VCPU and aVCPU personality instances, which is implemented by the management extensions in Dom0, as further discussed in Section 3.1.2.2. The outcome is that the hypervisor's scheduling methods operate *across two different and individually controlled scheduling domains*, one for the x86 cores managed by the hypervisor's underlying VCPU scheduler, the other for the the off-chip GPU managed by NVIDIA's proprietary drivers.

This can be done without the ability to control in detail how the NVIDIA driver allocates GPU resources.

**Asymmetric personality:** the accelerator personalities discussed above are fixed and do not change without using binary code rewriting or recompilation. However, personalities in the asymmetric systems described in Section 2.1 are more fluid. For example, consider a VCPU that first executes some image processing function that is extremely compute intensive and then switches to writing that (large) image out to the disk or network. The first processing phase maps well to a "fast" core [fast personality], while the second phase [IO or slow personality] maps well to a "slower" but possibly more power efficient core. In response, we introduce the notion of asymmetry-aware personality scheduling, which is responsible for mappping these subtler personalities to an asymmetric pool of hardware resources. Further and beyond making accelerators schedulable entities and managing single ISA performance asymmetries, the *personality-aware* or *HyVM hypervisor* uses additional methods to recognize asymmetric personalities and implement improvements in processor fungibility. The intent is to deal with functional asymmetry by dynamically varying where, when, and how different VM personalities are run, as shown in Figure 4.

(1) It uses the *fault and migrate* [65] method for dealing with runtime personality changes, in which a shared-ISA processor experiences a fault when an unsupported instruction is executed, whereupon the hypervisor migrates the sVCPU to a different core.

(2) It can also use *emulation*, for specific instructions (i.e., in the sVCPU case) or for different ISAs (i.e., in the GPGPU case). This is useful because when running in emulation mode, substantial data can be collected about the application, which can then be used for performance debugging. Further, recent performance results show that another reason for emulation is to efficiently use *all* of the processing resources on a platform. For example, a GPGPU code can often run just as well on a CPU [42, 68], e.g., in case of smaller data sizes or when the available GPGPUs are all busy. For the GPGPU case, HyVM implements this functionality via JIT translation of CUDA codes when running them on general-purpose

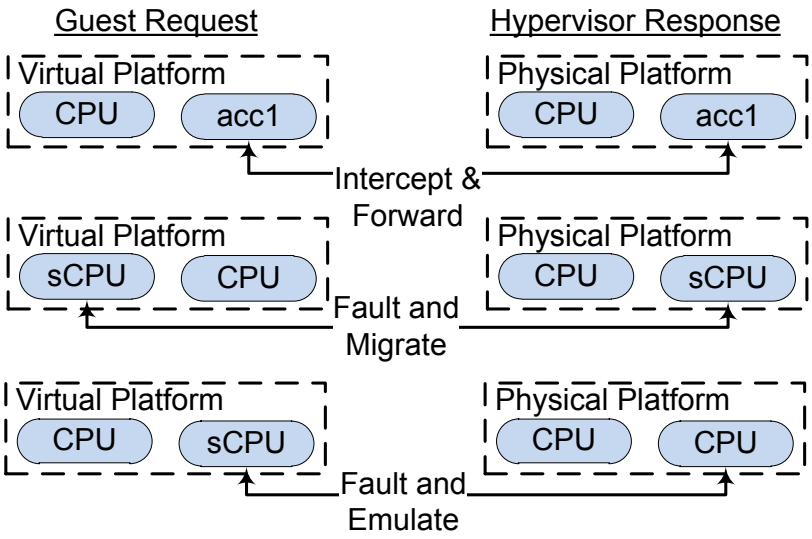



**Figure 4:** Mechanisms for personality change

CPUs, using the Ocelot [17] dynamic compilation framework. Unfortunately, the reverse action of running arbitrary CPU personalities on GPUs is currently not possible because threads running in a VM can access arbitrary portions of their process address spaces.

Figure 4 summarizes the multiple implementation options for dealing with runtime personality change and/or with multiple personalities associated with a single VM. We note that the idea of fault and migrate was first implemented in an operating system context in [64], but HyVM extends it to a virtualized system, so as to support guests that do not run specialized OS kernels.

#### *3.1.2.2 Personality Scheduling*

Our prior work with combined x86-GPGPU platforms [32] has demonstrated the need for diversity in the policies used for personality scheduling, to match the diverse application requirements and platform heterogeneities that will undoubtedly arise in systems like Amazon's HPC cloud [3] and other data centers going forward. We also argue that personality scheduling does not replace the techniques developed for managing accelerator or general purpose processors, as shown in Figure 3 where both target accelerators and general purpose cores have their own unique resource managers. This implies that personality

scheduling does not replace but instead, enhances the actions taken by schedulers operating in different scheduling domains. In the case of GPGPUs, such enhancements are realized as *coordination methods*, where the actions of different scheduling domains. coordination is carried out by controlling when and to which processing cores personalities are mapped and, if possible (i.e., if APIs exist or can be added), by interacting with the underlying schedulers in the respective domains (e.g., the Xen hypervisor's fair share scheduler and the Dom0 functionality that maps invocations made by guests to run on the GPGPU) to improve their joint behavior [31]. In the case of asymmetric hardware, the methods that match VCPUs to PCPUs interact with the underlying Xen fair share scheduler to maintain the 'virtual hardware' notion Xen maintains for the multiple VMs being run.

Stated more precisely, personality scheduling is implemented as follows. First and to realize the notion of a common pool of schedulable resources, personality scheduling makes sure that there are ready queues and scheduling functions associated with each processing element on the platform. Second, for heterogeneous hardware, the actual scheduling function may be part of the core hypervisor, as in case of the Xen credit scheduler managing general purpose cores, or it may be an interface to a proprietary runtime, such as the driver level scheduler provided by NVIDIA for running codes on GPGPUs. In that case, coordination methods – scheduling coordination – then logically implies modifications to the order in which schedulable units are placed into the cores' ready queues, and/or it may involve explicitly triggering some scheduler to pick up a given (a)VCPU, so as to co-schedule it with another VCPU running on a different (type of) core. Third, for asymmetric hardware, personality schedulers must go beyond what is done for the 'fixed personalities' destined for GPGPUs to also deal with runtime changes in VM personalities, meaning that they must be able to change a VCPU's mapping while it is running. An example is a VM with a workload that has complex, multi-phase behaviors like alternating serial vs. parallel phases that each benefit from certain specialized instructions. Fourth and to support such

dynamic changes, then, the HyVM approach must not only present a personality scheduling framework in which diverse scheduling methods can be implemented, but it must also provide APIs with which guests can notify the hypervisor about an impending personality change [31, 77] and/or runtime methods via which the hypervisor can detect such changes. Some changes can be detected easily, as when a VPCU repeatedly faults on certain instructions not supported by the core on which it currently runs, but others require more sophisticated runtime monitoring of guests and/or new dynamic code analysis techniques. We briefly comment on this below, but note that most of those topics, along with sophisticated policies for matching VCPUs to asymmetric PCPUs, remain subject for future research.

It is clear from the discussion above that in order to enable holistic management of the platform resources, the HyVM hypervisor must also (1) monitor the platform, (2) maintain estimates of the execution time of different or alternative VM personalities, and (3) assess the need for and potential benefits of runtime personality changes. Point-(1) implies gathering runtime information such as average load per core. Point-(2) can be estimated using various methods, such as (i) hints provided by the user, (ii) history information maintained by the schedulers, or (iii) runtime monitoring. Finally, point-(3) requires that the hypervisor not only support the appropriate methods for translation (i.e., the 'fault-and-migrate' or emulation methods described above), but also maintain the execution cost of such translations via profiles or continuous estimates, as well as any potential wait times incurred by a new personality due to the current load on the core of the target type. The generalization of such a personality scheduler for future systems is presented in Chapter 9.

# CHAPTER IV

# PEGASUS: COORDINATED SCHEDULING FOR VIRTUALIZED ACCELERATOR BASED SYSTEMS

As described in Chapters 1 and 2, systems with specialized processors like those used for accelerating computations, network processing, or cryptographic tasks [76, 106] have proven their utility in terms of higher performance and lower power consumption. This is not only causing tremendous growth in accelerator-based platforms, but it is also leading to the release of heterogeneous processors where x86-based cores and on-chip network or graphics accelerators [43, 97] form a common pool of resources. However, operating systems and virtualization platforms have not yet adjusted to these architectural trends. In particular, they continue to treat accelerators as secondary devices and focus scheduling and resource management on their general purpose processors, supported by vendors that shield developers from the complexities of accelerator hardware by 'hiding' it behind drivers that only expose higher level programming APIs [47, 77]. Unfortunately, technically, this implies that drivers rather than operating systems or hypervisors determine how accelerators are shared, which restricts scheduling policies and thus, the optimization criteria applied when using such heterogeneous systems.

A driver-based execution model can not only potentially hurt utilization, but also make it difficult for applications and systems to obtain desired benefits from the combined use of heterogeneous processing units. Consider, for instance, an advanced image processing service akin to HP's Snapfish [98] or Microsoft's PhotoSynth [71] applications, but offering additional computational services like complex image enhancement and watermarking, hosted in a data center. For such applications, the low latency responses desired by end users require the combined processing power of both general purpose and accelerator cores.

An example is the execution of sequences of operations like those that first identify spatial correlation or correspondence [99] between images prior to synthesizing them [71]. For these pipelined sets of tasks, some can efficiently run on multicore CPUs, whereas others can substantially benefit from acceleration [13, 68]. However, when they concurrently use both types of processing resources, low latency is attained only when different pipeline elements are appropriately co- or gang-scheduled onto both CPU and GPU cores. As shown later in this chapter, such co-scheduling is difficult to perform with current accelerators when used in consolidated data center settings. Further, it is hard to enforce fairness in accelerator use when the many clients in typical web applications cause multiple tasks to compete for both general purpose and accelerator resources,

*Pegasus* addresses the urgent need for systems support to smartly manage accelerators by enabling personality creation and management as discussed in Chapter 3. It does this by leveraging the new opportunities presented by increased adoption of virtualization technology in commercial, cloud computing [3], and even high performance infrastructures [59, 109]: *the Pegasus hypervisor extensions* (1) make accelerators into first class schedulable entities and (2) support scheduling methods that enable efficient use of both the general purpose and accelerator cores of heterogeneous hardware platforms. Specifically, for platforms comprised of x86 CPUs connected to NVIDIA GPUs, these extensions can be used to manage all of the platform's processing resources, to address the broad range of needs of GPGPU (general purpose computation on graphics processing units) applications, including the high throughput requirements of compute intensive web applications like the image processing code outlined above and the low latency requirements of computational finance [69] or similarly computationally intensive high performance codes. For high throughput, platform resources can be shared across many applications and/or clients. For low latency, resource management with such sharing also considers individual application requirements, including those of the inter-dependent pipeline-based codes employed for the financial and image processing applications.

The Pegasus hypervisor extensions described in Sections 4.2 and 4.4 do not give applications direct access to accelerators [77], nor do they hide them behind a virtual file system layer [11, 35]. Instead, similar to past work on self-virtualizing devices [84], Pegasus exposes to applications a virtual accelerator interface, and it supports existing GPGPU applications by making this interface identical to NVIDIA's CUDA programming API [30]. As a result, whenever a virtual machine attempts to use the accelerator by calling this API, control reverts to the hypervisor. This means, of course, that the hypervisor 'sees' the application's accelerator accesses, thereby getting an opportunity to regulate (schedule) them. A second step taken by Pegasus is to then explicitly *coordinate* how VMs use general purpose and accelerator resources. With the Xen implementation [14] of Pegasus describe in this chapter, this is done by explicitly scheduling guest VMs' accelerator accesses in Xen's Dom0, while at the same time controlling those VMs' use of general purpose processors, the latter exploiting Dom0's privileged access to the Xen hypervisor and its VM scheduler.

Pegasus elevates accelerators to first class schedulable citizens in a manner somewhat similar to the way it is done in the Helios operating system [75], which uses satellite kernels with standard interfaces for XScale-based IO cards. However, given the fast rate of technology development in accelerator chips, we consider it premature to impose a common abstraction across all possible heterogeneous processors. Instead, Pegasus uses a more loosely coupled approach in which it assumes systems to have different 'scheduling domains', each of which is adept at controlling its own set of resources, e.g., accelerator vs. general purpose cores. Pegasus scheduling, then, coordinates when and to what extent, VMs use the resources managed by these multiple scheduling domains. This approach leverages notions of 'cellular' hypervisor structures [27] or federated schedulers that have been shown useful in other contexts [54]. Concurrent use of both CPU and GPU resources is one class of coordination methods Pegasus implements, with other methods aimed at delivering both high performance and fairness in terms of VM usage of platform resources.

Pegasus relies on application developers or toolchains to identify the right target processors for different computational tasks and to generate such tasks with the appropriate instruction set architectures (ISAs). Further, its current implementation does not interact with tool chains or runtimes, but we recognize that such interactions could improve the effectiveness of its runtime methods for resource management [19]. An advantage derived from this lack of interaction, however, is that Pegasus does not depend on certain toolchains or runtimes, nor does it require internal information about accelerators [68]. As a result, Pegasus can operate with both 'closed' accelerators like NVIDIA GPUs and with 'open' ones like IBM Cell [33], and its approach can easily be extended to support other APIs like OpenCL [47].

Summarizing, the Pegasus hypervisor extensions make the following contributions:

**Accelerators as first class schedulable entities**—accelerators (accelerator physical CPUs or aPCPUs) can be managed as first class schedulable entities, i.e., they can be shared by multiple tasks, and task mappings to processors are dynamic, within the constraints imposed by the accelerator software stacks.

**Visible heterogeneity**—Pegasus respects the fact that aPCPUs differ in capabilities, have different modes of access, and sometimes use different ISAs. Rather than hiding these facts, Pegasus exposes heterogeneity to the applications and the guest virtual machines (VMs) that are capable of exploiting it.

**Diversity in scheduling**—accelerators are used in multiple ways, e.g., to speedup parallel codes, to increase throughput, or to improve a platform's power/performance properties. Pegasus addresses differing application needs by offering a diversity of methods for scheduling accelerator and general purpose resources, including co-scheduling for concurrency constraints.

**'Coordination' as the basis for resource management**—internally, accelerators use specialized execution environments with their own resource managers [33, 76]. Pegasus

uses *coordinated scheduling methods* to align accelerator resource usage with platform-level management. While coordination applies external controls to control the use of 'closed' accelerators, i.e., accelerators with resource managers that do not export coordination interfaces, it could interact more intimately with 'open' managers as per their internal scheduling methods.

**Novel scheduling methods**—current schedulers on parallel machines assume complete control over their underlying platforms' processing resources. In contrast, Pegasus recognizes and deals with heterogeneity not only in terms of differing resource capabilities, but also in terms of the diverse scheduling methods these resources may require, an example being the highly parallel internal scheduling used in GPGPUs. Pegasus coordination methods, therefore, differ from traditional co-scheduling in that they operate above underlying native techniques. Such meta-scheduling, therefore, seeks to influence the actions of underlying schedulers rather than replacing their functionality. This work proposes and evaluates new coordination methods that are geared to dealing with diverse resources, including CPUs vs. GPUs and multiple generations of the latter, yet at the same time, attempting to preserve desired virtual platform properties, including fair-sharing and prioritization.

The current Xen-based Pegasus prototype efficiently virtualizes NVIDIA GPUs, resulting in performance competitive with that of applications that have direct access to the GPU resources, as shown in Section 4.5. More importantly, when the GPGPU resources are shared by multiple guest VMs, online resource management becomes critical. This is evident from the performance benefits derived from the coordination policies described in Section 4.3, which range from 18% to 140% over base GPU driver scheduling. An extension to the current, fully functional, single-node Pegasus prototype will be deployed to a large-scale GPU-based cluster machine, called Keeneland, under construction at Oak Ridge National Labs [109], to further validate our approach and to better understand how to improve the federated scheduling infrastructures needed for future larger scale heterogeneous systems.

In the remaining chapter, Section 4.1 articulates the need for smart accelerator sharing. Section 4.2 outlines the Pegasus architecture. Section 4.3 describes its rich resource management methods. A discussion of scheduling policies is followed by implementation details in Section 4.4, and experimental evaluation in Section 4.5. Related work is in Section 4.6, followed by conclusions and future work.

## *4.1 Background*

This section offers additional motivation for the Pegasus approach on a heterogeneous multi-core platforms.

**Value in sharing resources**—Accelerator performance and usability (e.g., the increasing adoption of CUDA) are improving rapidly. However, even for today's platforms, the majority of applications do not occupy the entire accelerator [7, 45]. In consequence and despite continuing efforts to improve the performance of single accelerator applications [29], resource sharing is now supported in NVIDIA's Fermi architecture [76], IBM's Cell, and others. These facts are the prime drivers behind our decision to develop scheduling methods that can efficiently utilize both accelerator and general purpose cores. However, as stated earlier, for reasons of portability across different accelerators and accelerator generations, and to deal with their proprietary nature, Pegasus resource sharing across different VMs is implemented at a layer above the driver, leaving it up to the individual applications running in each VM to control and optimize their use of accelerator resources.

**Limitations of traditional device driver based solutions**—Typical accelerators have a sophisticated and often proprietary device driver layer, with an optional runtime. While these efficiently implement the computational and data interactions between accelerator and host cores [77], they lack support for efficient resource sharing. For example, first-come-first-serve issue of CUDA calls from 'applications-to-GPU' through a centralized NVIDIA-driver can lead to possibly detrimental call interleavings, which can cause high variances in call times and degradation in performance, as shown by measurements of the

NVIDIA driver in Section 4.5. Pegasus can avoid such issues and use a more favorable call order, by introducing and regulating time-shares for VMs to issue GPU-requests. This leads to significantly improved performance even for simple scheduling schemes.

## *4.2 Pegasus System Architecture*

Designed to generalize from current accelerator-based systems to future heterogeneous many-core platforms, Pegasus creates the logical view of computational resources shown in Figure 5. In this view, general purpose and accelerator tasks are schedulable entities mapped to VCPUs (virtual CPUs) characterized as general purpose or as 'accelerator'. Since both sets of processors can be scheduled independently, platform-wide scheduling, then, requires Pegasus to federate the platform's general purpose and accelerator schedulers. Federation is implemented by coordination methods that provide the serviced virtual machines with shares of physical processors based on the diverse policies described in Section 4.3. *Coordination is particularly important for closely coupled tasks running on both accelerator and general purpose cores*, as with the image processing application explained earlier. Figure 5 shows virtual machines running on either one or both types of processors, i.e., the CPUs and/or the accelerators. The figure also suggests the relative rarity of VMs running solely on accelerators (grayed out in the figure) in current systems. We segregate the privileged software components shown for the host and accelerator cores to acknowledge that the accelerator could have its own privileged runtime.

The following questions articulate the challenges in achieving the vision shown in Figure 5.

*How can heterogeneous resources be managed?* Hardware heterogeneity goes beyond varying compute speeds to include differing interconnect distances, different and possibly disjoint memory models, and potentially different or non-overlapping ISAs. This makes it difficult to assimilate these accelerators into one common platform. Exacerbating these hardware differences are software challenges, like those caused by the fact that there is

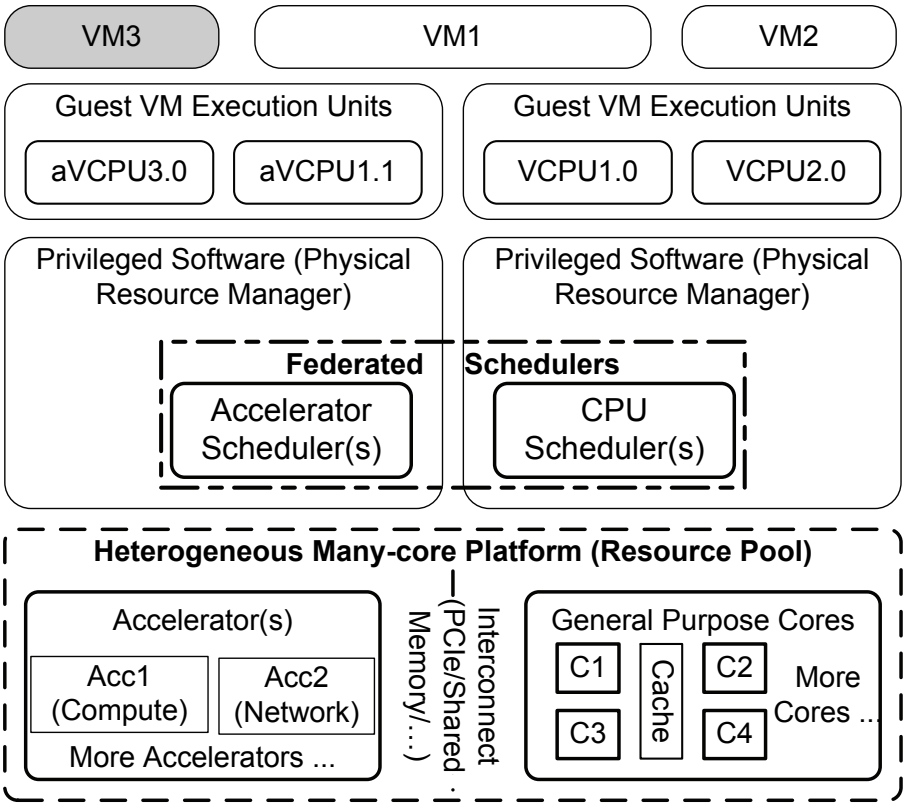



**Figure 5:** Logical view of Pegasus architecture

no general agreement about programming models and runtimes for accelerator-based systems [47, 77].

*Are there efficient methods to utilize heterogeneous resources?* The hypervisor has limited control over how the resources internal to closed accelerators are used, and whether sharing is possible in time, space, or both because there is no direct control over scheduler actions beyond the proprietary interfaces. The concrete question, then, is whether and to what extent the coordinated scheduling approach adopted by Pegasus can succeed.

Pegasus therefore allows schedulers to run resource allocation policies that offer diversity in how they maximize application performance and/or fairness in resource sharing.

### 4.2.1 Accelerator Virtualization - GViM Architecture

Figure 6 shows the system architecture of a virtualized GPGPU system termed GViM (GPU-accelerated Virtual Machines). The hardware platform consists of general purpose

cores (e.g., x86 cores) and specialized graphics accelerators – multiple NVIDIA GPUs in our prototype platform. Any number of VMs executing applications which require access to the GPU accelerators may be concurrently deployed in the system. The application components targeted for execution on the platform's GPU components are represented as kernels, and their deployment and invocation are supported by the CUDA API.

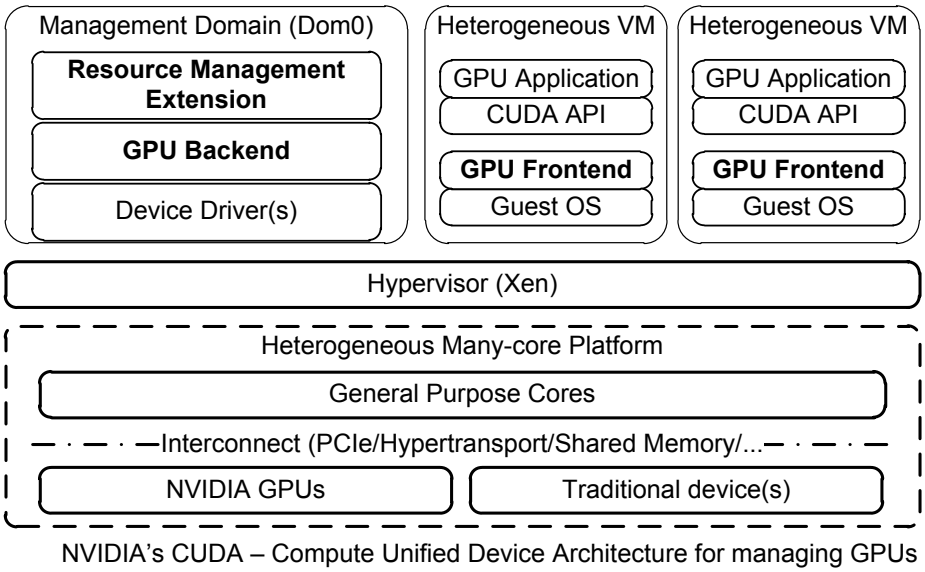



**Figure 6:** Virtualization of GPUs

The "split-driver model" depicted in Figure 6 delegates full control of the physical accelerators and devices to a management domain (i.e., Dom0 in Xen). This implies that all accesses to the GPU will be routed via the frontend/backend drivers through the management domain and that data moved between the GPU and the guest VM application may require multiple copies or page remapping operations [94, 95]. While the approach is sufficiently general to handle a range of devices without additional virtualization-related capabilities, the overheads associated with it are prohibitive for HPC applications. This is particularly true for the GPU accelerators due to the potentially large size of the input and output data of the kernel that can span many pages of contiguous memory. The GViM approach described here, adopts the split-driver model described above, but makes substantial enhancements to make it more suitable for the performance requirements of HPC

applications and the CUDA API on which many of them rely [111].

First, as CUDA has become an important API and programming model for high performance codes on GPU-accelerated manycore platforms, GViM virtualizes the GPU stack at the CUDA API level. While our choice of CUDA is a practical one that recognizes its substantial market penetration, it is also principled in that the level at which virtualization is done corresponds to that of other APIs used for accelerator interaction – IBM's ALF [38], originally developed for the Cell processors, the recently announced OpenCL [47], and many ongoing industry and academic efforts towards uniform APIs and access methods with associated languages and runtimes [19, 101, 67]. Through this approach, GViM provides for improved productivity, allowing developers to deal with familiar higher-level APIs, and for increased portability, hiding low level driver or architecture details from guest VMs. Furthermore, since CUDA's parallel programming model does not depend on the presence of graphics or other types of accelerators, CUDA kernels may also be deployed on the general purpose cores in the manycore platform, provided that appropriate binaries or translation tools exist. This gives GViM an additional level of flexibility to completely virtualize the heterogeneous platform resources. Our future work will focus on further enhancements that provide for and exploit such capabilities.

A key property of GViM is its ability to execute kernels with performance similar to that attained by VMs with direct access to accelerators. Toward this end, GViM implements efficient data movement between the guest VM's application using the kernel and the GPU running it. The enhancements to the standard front end/back end model for this purpose are similar to the VMM-bypass mechanisms supported for high performance interconnects such as InfiniBand, or developed for specialized programmable network interfaces [84]. Common to these approaches is that either the hardware device itself is capable of enforcing isolation and coordination across device accesses originating from multiple VMs (e.g., in the case of InfiniBand HCAs), or the hardware and software stack are 'open', i.e., programmable, and the device runtime is programmed to provide such functionality. Neither

one of these features is supported on our NVIDIA GPU accelerators or by their CUDA stack. NVIDIA's proprietary access model and binary device drivers make it a 'closed' accelerator architecture. In response, GViM offers 'VMM-bypass'-like functionality on the 'data-movement' path only, which means that all device accesses are still routed through the management domain, but then GViM uses lower-level memory management mechanisms to ensure that the kernels' input and output data is directly moved between the guest VM and the GPU. The result is the elimination of costly copy and remapping operations.

Building on this approach and acknowledging the current off-chip nature of accelerators, Pegasus assumes these hardware resources to be managed by both the hypervisor and Xen's 'Dom0' management (and driver) domain. Hence, Pegasus uses front end/back end split drivers [8] to mediate all accesses to GPUs connected via PCIe. Specifically, the requests for GPU usage issued by guest VMs (i.e., CUDA tasks) are contained in call buffers shared between guests and Dom0, as shown in Figure 7, using a separate buffer for each guest. Buffers are inspected by 'poller' threads that pick call packets from per-guest buffers and issue them to the actual CUDA runtime/driver resident in Dom0. These poller threads can be woken up whenever a domain has call requests waiting. This model of execution is well-matched with the ways in which guests use accelerators, typically wishing to utilize their computational capabilities for some time and with multiple calls.

For general purpose cores, a VCPU as the (virtual) CPU representation offered to a VM embodies the state representing the execution of the VM's threads/processes on physical CPUs (PCPUs). As a similar abstraction, Pegasus introduces the notion of an *accelerator VCPU (aVCPU)*, which embodies the VM's state concerning the execution of its calls to the accelerator. For the Xen/NVIDIA implementation, this abstraction is a combination of state allocated on the host and on the accelerator (i.e., Dom0 polling thread, CUDA calls, and driver context form the execution context while the data that is operated upon forms the data portion, when compared with the VCPUs). By introducing aVCPUs, Pegasus can then explicitly schedule them, just like their general purpose counterparts. Further, and as

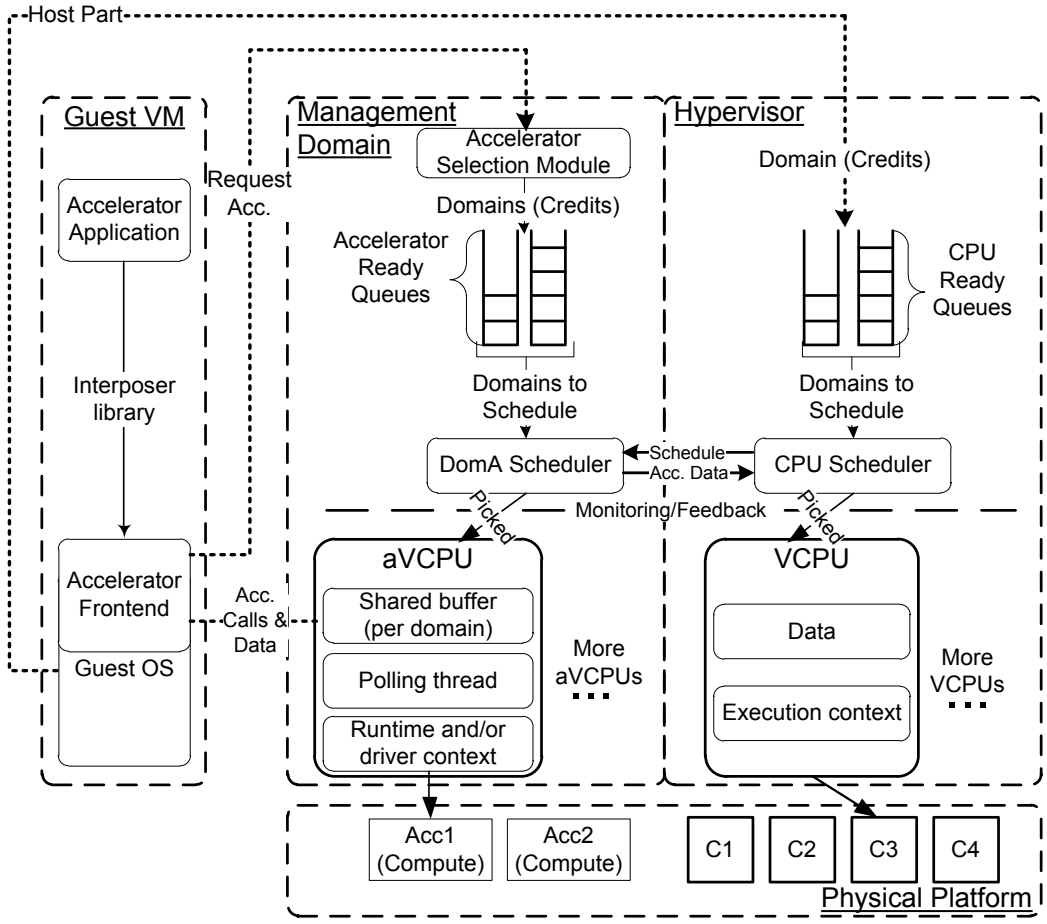



**Figure 7:** Logical view of the resource management framework in Pegasus

seen from Section 4.5, virtualization costs are negligible or low and with this API-based approach to virtualization, Pegasus leaves the use of resources on the accelerator hardware up to the application, ensures portability and independence from low-level changes in NVIDIA drivers and hardware.

### 4.2.2 Resource Management Framework

For VMs using both VCPUs and aVCPUs, resource management can explicitly track and schedule their joint use of both general purpose and accelerator resources. Technically, such management involves scheduling their VCPUs and aVCPUs to meet desired Service Level Agreement (SLA), concurrency constraints, and to ensure fairness in different guest VMs' resource usage.

For high performance, Pegasus distinguishes two phases in accelerator request scheduling. First, the **accelerator selection module** runs in the Accelerator Domain—which in our current implementation is Dom0—henceforth, called *DomA*. This module associates a domain, i.e., a guest VM, with an accelerator that has available resources, by placing the domain into an 'accelerator ready queue', as shown in Figure 7. Domains are selected from this queue when they are ready to issue requests. Second, it is only after this selection that actual usage requests are forwarded to, i.e., scheduled and run on, the selected accelerator. There are multiple reasons for this difference in accelerator vs. CPU scheduling. (1) An accelerator like the NVIDIA GPU has limited memory, and it associates a context with each 'user' (e.g., a thread) that locks some of the GPU's resources. (2) Memory swapping between host and accelerator memory over an interconnect like PCIe is expensive, which means that it is costly to dynamically change the context currently running on the GPU. In response, Pegasus GPU scheduling restricts the number of domains simultaneously scheduled on each accelerator and in addition, it permits each such domain to use the accelerator for some extensive time duration. The following parameters are used for accelerator selection.

*Accelerator profile and queue*—accelerators vary in terms of clock speed, memory size, in-out bandwidths and other such physical characteristics. These are static or hardware properties that can identify capability differences between various accelerators connected in the system. There also are dynamic properties like allocated memory, number of associated domains, etc., at any given time. This static and dynamic information is captured in an 'accelerator profile'. An 'accelerator weight' computed from this profile information determines current hardware capabilities and load characteristics for the accelerator. These weights are used to order accelerators in a priority queue maintained within the DomA Scheduler, termed as 'accelerator queue'. For example, the more an accelerator is used, the lower its weight becomes so that it does not get oversubscribed. The accelerator with the highest weight is the most capable and is the first to be considered when a domain requests

accelerator use.

*Domain profile*—domains may be more or less demanding of accelerator resources and more vs. less capable of using them. The 'domain profiles' maintained by Pegasus describe these differences, and they also quantitatively capture domain requirements. Concretely, the current implementation expects credit assignments [14] for each domain that gives it proportional access to the accelerator. Another example is to match the domain's expected memory requirements against the available memory on an accelerator (with CUDA, it is possible to determine this from application metadata). Since the execution properties of domains change over time, domain execution characteristics should be determined dynamically, which would then cause the runtime modification of a domain's accelerator credits and/or access privileges to accelerators. Automated methods for doing so, based on runtime monitoring, are subject of our future work, with initial ideas reported in [19]. This chapter lays the groundwork for such research: (1) we show coordination to be a fundamentally useful method for managing future heterogeneous systems, and (2) we demonstrate the importance of these runtime-based techniques and performance advantages derived from their use in a coordinated scheduling environment.

Once a domain has been associated with an accelerator, the **DomA Scheduler** in Figure 7 schedules execution of individual domain requests per accelerator by activating the corresponding domain's aVCPU. For all domains in its ready queue, the 'DomA Scheduler' has complete control over which domain's requests are submitted to the accelerator(s), and it can make such decisions in coordination with the hypervisor's VCPU scheduler, by exchanging relevant accelerator and schedule data. Scheduling in this second phase, can thus be enhanced by *coordinating* the actions of the hypervisor and DomA scheduler(s) present on the platform, as introduced in Figure 5. In addition, certain coordination policies can use the monitoring/feedback module, which currently tracks the average values of wait times for accelerator requests, the goal being to detect SLA violations for guest requests. Various policies supported by the DomA scheduler are described in the following section.

## *4.3 Scheduling Policies for Heterogeneity-aware Hypervisors*

Pegasus contributes its novel, federated, and heterogeneity-aware scheduling methods to the substantive body of past work in resource management. The policies described below, and implemented by the DomA scheduler, are categorized based on their level of interaction with the hypervisor's scheduler. They range from simple and easily implemented schemes offering basic scheduling properties to coordination-based policies that exploit information sharing between the hypervisor and accelerator subsystems. *Policies are designed to demonstrate the range of achievable coordination between the two scheduler subsystems and the benefits seen by such coordination for various workloads*. The specific property offered by each policy is indicated in square brackets.

### 4.3.1 Hypervisor Independent Policies

The simplest methods do not support scheduler federation, limiting their scheduling logic to DomA.

**No scheduling in backend (None) [first-come-first-serve]**—provides base functionality that assigns domains to accelerators in a round robin manner, but relies on NVIDIA's runtime/driver layer to handle all request scheduling. DomA scheduler plays no role in domain request scheduling. This serves as our baseline.

**AccCredit (AccC) [proportional fair-share]**—recognizing that domains differ in terms of their desire and ability to use accelerators, accelerator credits are associated with each domain, based on which different domains are polled for different time periods. This makes the time given to a guest proportional to how much it desires to use the accelerator, as apparent in the pseudo-code shown in Algorithm 1, where the requests from the domain at the head of the queue are handled until it finishes its awarded number of ticks. For instance, with credit assignments (Dom1,1024), (Dom2,512), (Dom3,256), and (Dom4,512), the number of ticks will be 4, 2, 1, and 2, respectively.

**Algorithm 1**: Simplified Representation of Scheduling Data and Functions for Credit-based Schemes

```
/* D = Domain being considered */
/* X = Domain cpu or accelerator credits */
/* T = Scheduler timer period */
/* Tc = Ticks assigned to next D */
/* Tm = maximum ticks D gets based on X */
Data: Ready queue RQ_A of domains (D)
/* RQ is ordered by X */
Data: Accelerator queue AccQ of accelerators
/* AccQ is ordered by accelerator weight */

InsertDomainforScheduling(D)
  if D not in RQ_A then
      T_c ← 1, T_m ← X / X_min
      A ← PickAccelerator(AccQ,D)
      InsertDomainInRQ_CreditSorted(RQ_A,D)
  else
      /* D already in some RQ_A  */
      if ContextEstablished then
          T_c ← T_m
      else
          T_c ← 1


DomASchedule(RQ_A)
  InsertDomainforScheduling(Curr_Dom)
  D ← RemoveHeadandAdvance(RQ_A)
  Set D's timer period to T_c; Curr_dom ← D
```

Because the accelerators used with Pegasus require their applications to explicitly allocate and free accelerator state, it is easy to determine whether or not a domain currently has context (state) established on an accelerator. The DomA scheduler, therefore, interprets a domain in a ContextEstablished state as one that is actively using the accelerator. When in a NoContextEstablished state, a minimum time tick (1) is assigned to the domain for the next scheduling cycle (see Algorithm 1).

### 4.3.2 Hypervisor Controlled Policy

The rationale behind coordinating VCPUs and aVCPUs is that the overall execution time of an application (comprised of both host and accelerator portions) can be reduced if its communicating host and accelerator tasks are scheduled at the same time. We implement one such method described next.

**Algorithm 2:** Simplified Representation of CoSched and AugC Schemes

```
/* RQ_cpu=Per CPU ready q in hypervisor */
/* HS=VCPU-PCPU schedule for next period */
/* X = domain credits */

HypeSchedule(RQ_cpu)
  Pick VCPUs for all PCPUs in system
  ∀D, AugCredit_D = RemainingCredit
  Pass HS to DomA scheduler

DomACoSchedule(RQ_A, HS)
  /* To handle #cpus > #accelerators */
  ∀D ∈ (RQ_A ⋂ HS)
    Pick D with highest X
  if D = null then
      /* To improve GPU utilization */
      Pick D with highest X in RQ_A

DomAAugSchedule(RQ_A, HS)
  foreach D ∈ RQ_A do
    Pick D with highest (AugCredit + X)
```

**Strict co-scheduling (CoSched) [latency reduction by occasional unfairness]**—an alternative to the accelerator-centric policies shown above, this policy gives complete control over scheduling to the hypervisor. Here, accelerator cores are treated as slaves to host cores, so that VCPUs and aVCPUs are scheduled at the same time. This policy works particularly well for latency-sensitive workloads like certain financial processing codes [69] or barrier-rich parallel applications. It is implemented by permitting the hypervisor scheduler to control how DomA schedules aVCPUs, as shown in Algorithm 2. For 'singular VCPUs', i.e., those without associated aVCPUs, scheduling reverts to using a standard credit-based scheme.

### 4.3.3 Hypervisor Coordinated Policies

A known issue with co-scheduling is potential unfairness. The following methods have the hypervisor actively participate in making scheduling decisions rather than governing them:

**Augmented credit-based scheme (AugC) [throughput improvement by temporary**

**credit boost]**—going beyond the proportionality approach in AccC, this policy uses active coordination between the DomA scheduler and hypervisor (Xen) scheduler in an attempt to better co-schedule domains on a CPU and GPU. To enable coscheduling, the Xen credit-based scheduler provides to the DomA scheduler, as a hint, its CPU schedule for the upcoming period, with remaining credits for all domains in the schedule as shown in Algorithm 2. The DomA scheduler uses this schedule to add temporary credits to the corresponding domains in its list (i.e., to those that have been scheduled for the next CPU time period). This boosts the credits of those domains that have their VCPUs selected by CPU scheduling, thus increasing their chances for getting scheduled on the corresponding GPU. While this effectively co-schedules these domains' CPU and GPU tasks, the DomA scheduler retains complete control over its actions; no domain with high accelerator credits is denied its eventual turn due to this temporary boost.

**SLA feedback to meet QoS requirements (SLAF) [feedback-based proportional fair-share]**—this is an adaptation of the AccC scheme as shown in Algorithm 1, with feedback control. (1) We start with an SLA defined for a domain (statically profiled) as the expected accelerator utilization—e.g., 0.5sec every second. (2) As shown in Algorithm 1, once the domain moves to a ContextEstablished state, it is polled, and its requests are handled for its assigned duration. In addition, a sum of domain poll time is maintained. (3) Ever so often, all domains associated with an accelerator are scanned for possible SLA violations. Domains with violations are given extra time ticks to compensate, one per scheduling cycle. (4) In high load conditions, there is a trigger that increases accelerator load in order to avoid new domain requests, which in the worst case, forces domains with comparatively low credits to wait longer to get compensated for violations seen by higher credit domains.

For generality in scheduling, we have also implemented: (1) Round robin (RR) [fair-share] which is hypervisor independent, and (2) XenoCredit (XC) [proportional fair-share] which is similar to AccC except it depends on CPU credits assigned to the corresponding

VM, making it a hypervisor coordinated policy.

## *4.4 System Implementation*

The current Pegasus implementation operates with Xen and NVIDIA GPUs. As a result, resource management policies are implemented within the management framework (Section 4.2.2) run in DomA (i.e., Dom0 in the current implementation), as shown in Figure 7.

### 4.4.1 GViM Implementation

This section describes how GViM virtualizes an NVIDIA-based GPGPU platform. The NVIDIA accelerator supports the CUDA higher end parallel execution model with reported speedups ranging from 18x to 140x compared to general purpose CPUs. Virtualizing it, however, entails considerable complexity due to its proprietary access model and binary device drivers (i.e., its 'closed' architecture). We virtualize this accelerator on a hardware platform comprised of an x86-based multicore node with multiple accelerators attached via PCIe devices, using the Xen hypervisor and Linux as the target guest OS. The platform is designed to emulate future heterogeneous manycore chips comprised of both general and special purpose processors.

#### *4.4.1.1 Design*

With the Xen hypervisor, a GPU attached to the host system must run its drivers in a privileged domain that can directly access the hardware. An example of a privileged domain is Xen's management domain (henceforth referred to as 'Dom0'). This privileged domain, therefore, must also implement suitable memory management and communication methods for efficient sharing of the GPU by multiple guest VMs. In the rest of this chapter, we assume that the drivers are run in Dom0 but note that they could also be run in a separate "driver domain" [25].

Figure 8 shows the implementation components involved in virtualizing (and sharing) the GPU for access by multiple VMs. To attain high performance, GViM adapts Xen's

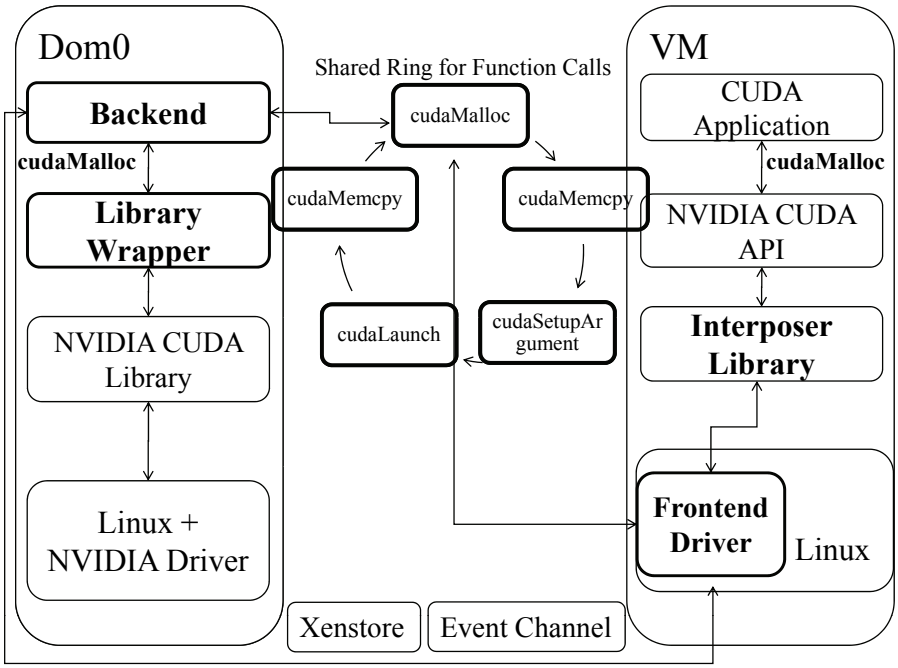



**Figure 8:** Virtualization components for GPU

split driver model in multiple ways, the most important one being our mechanisms for data bypass, as described in further detail below.

The following adaptations exist on the guest side, as depicted on the right half of Figure 8, which shows a VM and the software layers being used for GPU access:

1. The GPU application uses the CUDA API – as explained in Section 3.1.2, GViM permits users to run any arbitrary CUDA-based application in a VM.

2. The Interposer Library provides CUDA access – In a non-virtualized environment, GPU applications written with the CUDA API rely on libraries available with the CUDA SDK. These libraries perform required checks and pass parameters to the lower level driver that triggers execution on a GPU. Since the source code of the library and driver are 'closed', the interposer library running in the guest VM intercepts CUDA calls made by an application, collects arguments, packs them into a CUDA call packet, and sends the packet to the frontend driver described below. GViM thus maintains the abstraction level required for broad application use. The

library currently implements all function calls synchronously and is capable of managing multiple application threads.

3. Frontend driver - The Frontend driver manages the connections between the guest VM and Dom0. It uses Xenbus and Xenstore [14] to establish event channels between both domains, receiving call packets from the interposer library, sending these requests to the backend for execution over a shared call buffer (or shared ring in Xen terminology), and relaying responses back to the interposer library. GViM's implementation localizes all changes to the guest OS within the frontend driver, which can be loaded as a kernel module.

Function calls are carried out by several components in Dom0, which are described next and shown in the left half of Figure 8:

1. The Backend mediates all accesses to the GPU – located in Dom0, the backend is responsible for executing CUDA calls received from the frontend and for returning the execution results. It notifies the guest once the call has executed, and the result is passed via a shared ring. It is implemented as a user-level module for easier integration with the user-level CUDA libraries and to avoid additional runtime overhead due to accesses to userspace CUDA memory.

2. The Library Wrapper functions convert the call packets received from the frontend into function calls – the wrapper functions unpack these packets and execute the appropriate CUDA functions.

3. The NVIDIA CUDA library and driver are provided by NVIDIA – they are the components that interact with the actual device.

Jointly, these components enable a guest virtual machine to access any number of GPUs available in the system. The stack shown on the VM-side in Figure 8 is replicated in every guest in the system that wishes to access the GPU, while the single Dom0 stack is

responsible for managing multiple guest domains as well as multiple GPUs, if available. In Section 4.3, we further discuss as 'management extension' the interaction between the backend and the scheduling of requests made by guests.

#### *4.4.1.2 Memory Allocation and Sharing*

Many HPC applications using GPUs have large amounts of input and/or output data. Xen does not natively provide efficient (i.e., non copy-based) support for large data sharing between guest VMs and Dom0. Prior efforts to improve IO performance [115, 94] for network and block devices do not address sharing large numbers of pages with contiguous virtual addresses between a guest domain and a Dom0 backend running as a user level process, as required for GViM applications and as shown in Figure 9, which depicts with dotted lines the memory-related interactions in GViM. The dark solid arrows indicate the path of a CUDA call (refer to Figure 8 for detail). The arrows between guest application, Malloc memory, and Frontend memory allocator (via Frontend) are explained below.

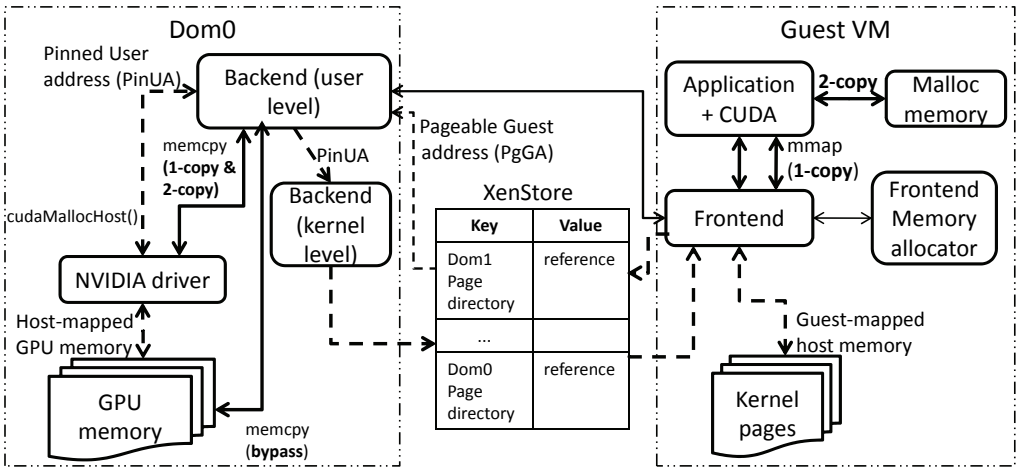



**Figure 9:** Memory management in GViM

In a non-virtualized environment, a memcpy operation invoked by a CUDA application has to go through one copy of data between host memory and GPU memory, managed by the NVIDIA driver. This can be avoided whenever possible/desirable by making a call to cudaMallocHost() which tries to return a pointer to host-mapped GPU memory,

as shown in the left part of Figure 9. This is the pinned memory case and implies zero-copy of data. To understand the memory management in GViM in greater detail, it is important to understand the different memory *kinds* that can be allocated by the user for data movements:

1. 2-copy – A user CUDA application running in the guest VM can allocate memory for data buffers using malloc. The buffer is then resident in the user virtual address space; this makes it necessary to copy that data into a temporary kernel buffer before passing it on to Dom0 for some CUDA operation. However, it is possible to share the kernel buffer in advance with the backend and eliminate the copy from the guest kernel to the backend. The next step is a copy from host memory to GPU memory, as described above. Since the data has to go through two copies, this is the 2-copy solution.

2. 1-copy – From the previous solution, if we remove the guest 'user to kernel' copy before passing the call to the backend, we can reduce overheads, particularly for larger data movements. This is done by letting the user call mmap() into the frontend driver to get memory for its buffer instead of calling malloc. To avoid changing applications, the mmap call has been wrapped within cudaMallocHost() implementation of the interposer library. This memory is pre-allocated by the frontend, shared with the Backend using Xenstore, and managed using the Frontend memory allocator. In order to allow a contiguous view of the memory at guest user level and at the Backend, the individual page numbers from the Frontend allocated buffer are loaded in a page directory structure that is shared with the Backend at frontend load time and is remapped by the Backend. We have thus, eliminated an extra level of copy potentially caused by virtualization.

3. Bypass – The ideal situation is a zero-copy data bypass whenever possible, as seen from Figure 12 in Section 4.5. Since the GPU memory is managed entirely by the

'closed' driver, we propose to let our Backend make a call to cudaMallocHost() when the system starts and map it through the kernel level module that can be loaded on its behalf into Dom0's kernel address space. Portions of this region can be mapped into individual guests and further used to move data to and from the card, i.e., eliminating the 'host to GPU' data movement shown in Figure 9. Inspite of its obvious performance benefit, the bypass approach limits the data sizes a VM can exchange with the GPU to the amount of available memory in the VM's partition of the driver memory. Therefore the 1-copy mechanism described above remains useful.

### 4.4.2 Scheduling

**Discovering GPUs and guest domains**: the management framework discovers all of the GPUs present in the system, assembles their static profiles using *cudaGetDeviceProperties()* [77], and registers them with the Pegasus hypervisor scheduling extensions. When new guest domains are created, Xen adds them to its hypervisor scheduling queue. Our management framework, in turn, discovers them by monitoring XenStore.

The scheduling policies RR, AccC, XC, and SLAF are implemented using timer signals, with one tick interval equal to the hypervisor's CPU scheduling timer interval. There is one timer handler or scheduler for each GPU, just like there is one scheduling timer interrupt per CPU, and this function picks the next domain to run from corresponding GPU's ready queue, as shown in Algorithm 1. AugC and CoSched use a thread in the backend that performs scheduling for each GPU by checking the latest schedule information provided by the hypervisor, as described in Section 4.3. It then sleeps for one timer interval. The per domain pollers are woken up or put to sleep by scheduling function(s), using real time signals with unique values assigned to each domain. This restricts the maximum number of domains supported by the backend to the Dom0 operating system imposed limit, but results in bounded/prioritized signal delivery times.

Two entirely different scheduling domains, i.e., DomA and the hypervisor, control the

two different kinds of processing units, i.e., GPUs and x86 cores. This poses several implementation challenges for the AugC and CoSched policies such as: (1) What data needs to be shared between extensions and the hypervisor scheduler and what additional actions to take, if any, in the hypervisor scheduler, given that this scheduler is in the critical path for the entire system? (2) How do we manage the differences and drifts in these respective schedulers' time periods?

Concerning (1), the current implementation extends the hypervisor scheduler to simply have it share its VCPU-PCPU schedule with the DomA scheduler, which then uses this schedule to find the right VM candidates for scheduling. Concerning (2), there can be a noticeable timing gap between when decisions are made and then enacted by the hypervisor scheduler vs. the DomA extensions. The resulting delay as to when or how soon a VCPU and an aVCPU from same domain are co-scheduled can be reduced with better control over the use of GPU resources. Since NVIDIA drivers do not offer such control, there is notable variation in co-scheduling. Our current remedial solution is to have each aVCPU be executed for 'some time', i.e., to run multiple CUDA call requests, rather than scheduling aVCPUs at a per CUDA call granularity, thereby increasing the possible overlap time with its 'co-related' VCPU. This does not solve the problem, but it mitigates the effects of imprecision, particularly for longer running workloads.

## *4.5 Experimental Evaluation*

Key contributions of Pegasus are (1) accelerators as first class schedulable entities and (2) coordinated scheduling to provide applications with the high levels of performance sought by use of heterogeneous processing resources. This section first shows that the Pegasus way of virtualizing accelerators is efficient, next demonstrates the importance of coordinated resource management, and finally, presents a number of interesting insights about how diverse coordination (i.e., scheduling) policies can be used to address workload diversity.

**Testbed:** All experimental evaluations are conducted on a system comprised of (1) a 2.5GHz Xeon quad-core processor with 3GB memory and (2) an NVIDIA 9800 GTX card with 2 GPUs and the v169.09 GPU driver. The Xen 3.2.1 hypervisor and the 2.6.18 Linux kernel are used in Dom0 and guest domains. Guest domains use 512MB memory and 1 VCPU each, the latter pinned to certain physical cores, depending on the experiments being conducted.

### 4.5.1 Benchmarks and Applications

Pegasus is evaluated with an extensive set of benchmarks and with emulations of more complex computationally expensive enterprise codes like the web-based image processing application mentioned earlier. Benchmarks include (1) parallel codes requiring low levels of deviation for highly synchronous execution, and (2) throughput-intensive codes. A complete listing appears in Table 1, identifying them as belonging to either the parboil benchmark suite [90] or the CUDA SDK 1.1. Benchmark-based performance studies go beyond running individual codes to using representative code mixes that have varying needs and differences in behavior due to different dataset sizes, data transfer times, iteration complexity, and numbers of iterations executed for certain computations. The latter two are a good measure of GPU 'kernel' size and the degree of coupling between CPUs orchestrating accelerator use and the GPUs running these kernels respectively. Depending on their outputs and the number of CUDA calls made, (1) throughput-sensitive benchmarks are MC, BOp, PI, (2) latency-sensitive benchmarks include FWT, and scientific, and (3) some benchmarks are both, e.g., BS, CP. A benchmark is throughput-sensitive when its performance is best evaluated as the number of some quantity processed or calculated per second, and a benchmark is latency-sensitive when it makes frequent CUDA calls and its execution time is sensitive to potential virtualization overhead and/or delays or 'noise' in accelerator scheduling. The image processing application, termed PicSer, emulates web codes like PhotoSynth. BlackScholes represents financial codes like those run by option

**Table 1:** Summary of Benchmarks

| Category | Source | Benchmarks |
|---|---|---|
| Financial | SDK | Binomial(BOp), BlackScholes(BS), Monte-Carlo(MC) |
| Media processing | SDK or parboil | ProcessImage(PI)=matrix multiply+DXTC, MRIQ, FastWalshTransform(FWT) |
| Scientific | parboil | CP, TPACF, RPES |

trading companies [69].

### 4.5.2 GPGPU Virtualization

Virtualization overheads when using Pegasus are depicted in Figures 10(a)–(d), using the benchmarks listed in Table 1. Results show the overhead (or speedup) when running the benchmark in question in a VM vs. when running it in Dom0. The overhead is calculated as the time it takes the benchmark to run in a VM divided by the time to run it in Dom0. We show the overhead (or speedup) for the average total execution time (Total Time) and the average time for CUDA calls (Cuda Time) across 50 runs of each benchmark. Cuda Time is calculated as the time to execute all CUDA calls within the application. Running the benchmark in Dom0 is equivalent to running it in a non-virtualized setting. For the 1VM numbers in Figure 10(a) and (c), all four cores are enabled, and to avoid scheduler interactions, Dom0 and the VM are pinned on separate cores. The experiments reported in Figure 10(b) have only 1 core enabled and the execution times are not averaged over multiple runs, with a backend restart for every run. This is done for reasons explained next. All cases use an equal number of physical GPUs, and Dom0 tests are run with as many cores as the Dom0–1VM case.

An interesting observation about these results is that sometimes, it is better to use virtualized rather than non-virtualized accelerators. This is because (1) the Pegasus virtualization software can benefit from the concurrency seen from using different cores for the guest vs. Dom0 domains, and (2) further advantages are derived from additional caching of data due to a constantly running—in Dom0—backend process and NVIDIA driver. This

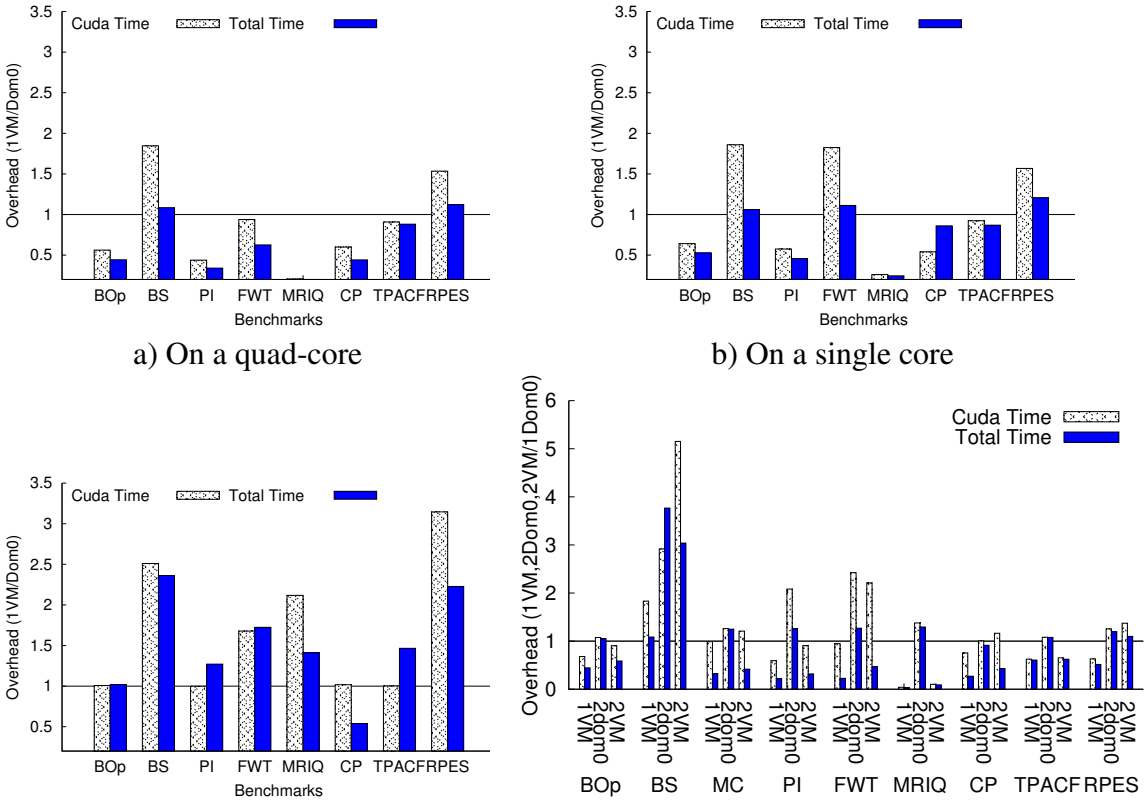



**Figure 10:** Evaluation of GPU virtualization overhead (lower is better)

is confirmed in Figure 10(b), which shows higher overheads when the backend is stopped before every run, wiping out any driver cache information. Also of interest is the speedup seen by say, BOp or PI vs. the performance seen by say, BS or RPES, in Figure 10(a). This is due to an increase in the number of calls per application, seen in BOp/PI vs. BS/RPES, emphasizing the virtualization overhead added to each executed CUDA call. In these cases, the benefits from caching and the presence of multiple cores are outweighed by the per call overhead multiplied by the number of calls made.

*Comparison of individual function call timings* to study virtualization overhead – Figure 11 shows the difference in execution time at a function call level between a virtualized guest and our base case of Dom0. These functions represent the most commonly used CUDA calls in GPU applications. The number of bytes transferred per CUDA call each way without data buffers is about 80Bytes which we refer to as the standard packet size

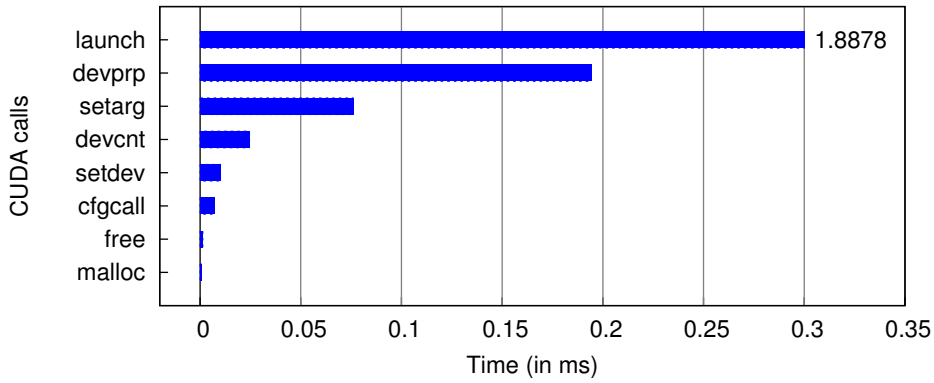



**Figure 11:** CUDA Calls - Execution Time Difference

(SPS). With the exception of devprp, setarg, launch, and memcpy (dealt with later), most calls exchange standard packet data both ways. Most commonly used CUDA calls do not see more than a 0.07msec increase in execution time except devprp (retrieving device properties) which is called once at the beginning and launch (cudaLaunch() for a GPU kernel). A typical sequence of operations for compute kernel execution on a GPU is a) configuring a call with appropriate thread and block sizes on the GPU, b) setting up arguments and c) launching the kernel. While these calls execute individually in a non-virtualized environment, we combine them together with launch in the backend due to requirements imposed by the driver level API. This leads to a higher launch time but keeps the overhead for configuring a call smaller.

*Impact of input data sizes* – The cost of memcpy (memory copy) varies with the amount of data transferred. These results become very important when applications transfer much larger data sizes (possibly multiple times during the application execution), as discussed in Section 4.4.1.

Figure 12 shows the bandwidth in MB/sec into (upper half of figure) and out (lower half) of the GPU for the guest VM, as well as Dom0 for the cases discussed in Section 4.4.1. The results (the Y-axis in figure does not start from 0) are obtained by running the GPU bandwidth test from the CUDA SDK. Dom0 pageable and pinned in the figure refer to the 1-copy and bypass options respectively for applications running in Dom0. As seen from

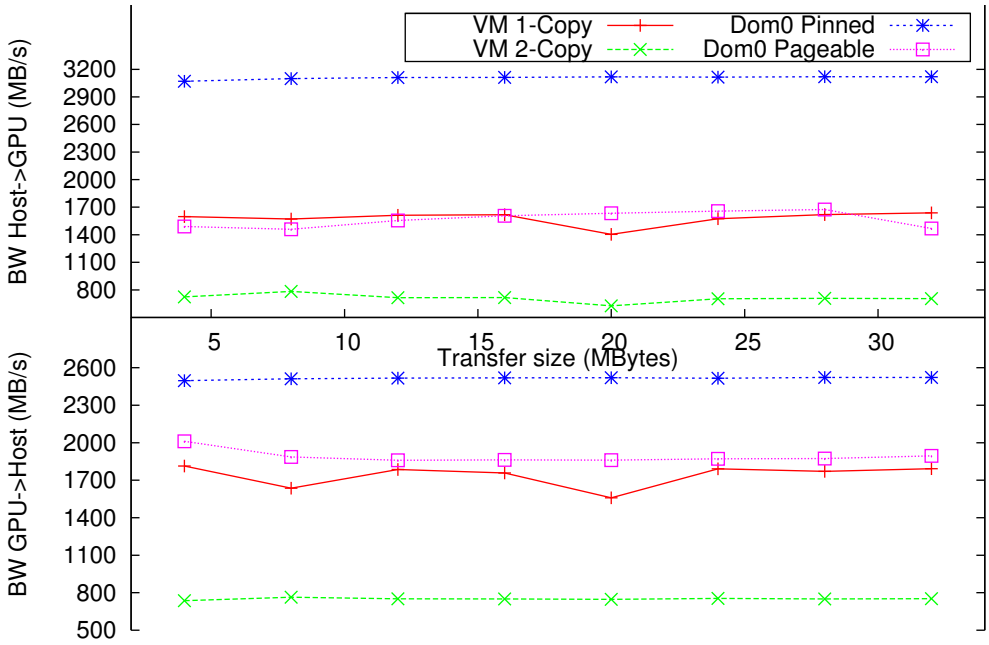



**Figure 12:** GPU bandwidth for memory copy operations

the graphs, our 1-copy solution achieves almost as much bandwidth as Dom0 Pageable. Since most applications are written to use pageable memory, this is a very good option for even large benchmarks being run in guest VM (as long as it has sufficient memory). The Dom0 pinned case shows much higher bandwidth, and we are working towards this bypass solution for our virtualized guests, as well.

### 4.5.3 Resource Management

The Pegasus framework for scheduling coordination makes it possible to implement diverse policies to meet different application needs. Consequently, we use multiple metrics to evaluate the policies described in Section 4.3. They include (1) throughput (*Quantity/sec*) for throughput-sensitive applications, (2) work done (*Quantity work/sec*) (which is the sum of the calculations done over all runs divided by the total time taken), and/or (3) per call latency (Latency) observed for CUDA calls (latency is reported including the CUDA function execution time to account for the fact that we cannot control how the driver orders the requests it receives from Pegasus).

**Experimental Methodology**: To reduce scheduling interference from the guest OS,

each VM runs only a single benchmark. Each sample set of measurements, then, involves launching the required number of VMs, each of which repeatedly runs its benchmark. To evaluate accelerator sharing, experiments use 2, 3, or 4 domains, which translates to configurations with no GPU/CPU sharing, sharing of one GPU and one CPU by two of the three domains, and sharing of two CPUs and both GPUs by pairs of domains, respectively. In all experiments, Dom1 and Dom3 share a CPU as well as a GPU, and so do Dom2 and Dom4, when present. Further, to avoid non-deterministic behavior due to actions taken by the hypervisor scheduler, and to deal with the limited numbers of cores and GPGPUs available on our experimental platform, we pin the domain VCPUs to certain CPUs, depending on the experiment scenario. These CPUs are chosen based on the workload distribution across CPUs (including backend threads in Dom0) and the concurrency requirements of VCPU and aVCPU from the same domain (simply put, VCPU from a domain and the polling thread forming its aVCPU cannot be co-scheduled if they are bound to the same CPU).

The results shown in this section focus on the BS benchmark, because of (1) its closeness to real world financial workloads, (2) its tunable iteration count argument that varies its CPU-GPU coupling and can highlight the benefits of coordination, (3) its easily varied data sizes and hence GPU computation complexity, and (4) its throughput as well as latency sensitive nature. Additional reported results are for benchmarks like PicSer, CP and FWT in order to highlight specific interesting/different cases, like those for applications with low degrees of coupling or with high latency sensitivity. For experiments that assign equal credits to all domains, we do not plot RR and AccC, since they are equivalent to XC. Also, we do not show AccC if accelerator credits are equal to Xen credits.

Observations at an early stage of experimentation showed that the CUDA driver introduces substantial variations in execution time when a GPU is shared by multiple applications (shown by the NoVirt graph in Figure 18). This caused us to use a large sample size of 50 runs per benchmark per domain, and we report either the h-spread[113] or work done which is the sum of total output divided by total elapsed time over those multiple runs. For

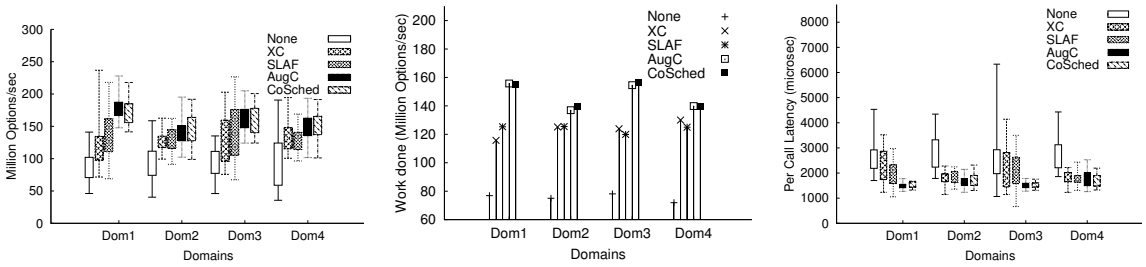
throughput and latency based experiments, we report values for 85% of the runs from the execution set, which prunes some outliers that can greatly skew results and thus, hide the important insights from a particular experiment. These outliers are typically introduced by (1) a serial launch of domains causing the first few readings to show non-shared timings for certain domains, and (2) some domains completing their runs earlier due to higher priority and/or because the launch pattern causes the last few readings for the remaining domains to again be during the unshared period. Hence, all throughput and latency graphs represent the distribution of values across the runs, with a box in the graph representing 50% of the samples around the median (or h-spread) and the lower and upper whiskers encapsulating 85% of the readings, with the minimum and maximum quantities as delimiters. It is difficult, however, to synchronize the launches of domains' GPU kernels with the execution of their threads on CPUs, leading to different orderings of CUDA calls in each run. Hence, to show cumulative performance over the entire experiment, for some experimental results, we also show the 'work done' over all of the runs.

***Scheduling is needed when sharing accelerators***: Figure 10(c) shows the overhead of sharing the GPU when applications are run both in Dom0 and in virtualized guests. In the figure, the 1VM quantities refer to overhead (or speedup) seen by a benchmark running in 1VM vs. when it is run nonvirtualized in Dom0. 2dom0 and 2VM values are similarly normalized with respect to the Dom0 values. 2dom0 values indicate execution times observed for a benchmark when it shares a GPU running in Dom0, i.e., in the absence of GPU virtualization, and 2VM values indicate similar values when run in two guest VMs sharing the GPU. For the 2VM case, the Backend implements RR, a scheduling policy that is completely fair to both VMs, and their CPU credits are set to 256 for equal CPU sharing. These measurements show that (1) while the performance seen by applications suffers from sharing (due to reduced accelerator access), (2) a clear benefit is derived for most benchmarks from using even a simple scheduling method for accelerator access. This is evident from the virtualized case that uses a round robin scheduler, which shows better

performance compared with the nonvirtualized runs in Dom0 for most benchmarks, particularly the ones with lower numbers of CUDA call invocations. This shows that *scheduling is important to reduce contention in the NVIDIA driver* and thus helps minimize the resulting performance degradation. Measurements report Cuda Time and Total Time, which is the metric used in Figures 10(a)–(b).

We speculate that sharing overheads could be reduced further if Pegasus was given more control over the way GPU resources are used. Additional benefits may arise from improved hardware support for sharing the accelerator, as expected for future NVIDIA hardware [76].

***Coordination can improve performance***: With encouraging results from the simple RR scheduler, we next experiment with the more sophisticated policies described in Section 4.3. In particular, we use BlackScholes (outputs options and hence its throughput is given by Options/sec) which, with more than 512 compute kernel launches and a large number of CUDA calls, has a high degree of CPU-GPU coupling. This motivates us to also report the latency numbers seen by BS.

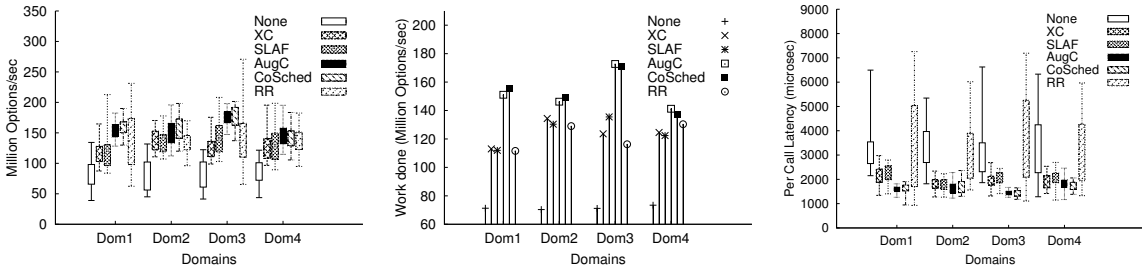



a) Throughput distribution b) Average work done c) Latency distribution

**Figure 13:** Performance of different scheduling schemes [BS]—equal credits for four guests

An important insight from these experiments is that coordination in scheduling is particularly important for tightly coupled codes, as demonstrated by the fact that our base case, None, shows large variations and worse overall performance, whereas AugC and CoSched show the best performance due to their higher degrees of coordination. Figures 13(a)–(c)

show that these policies perform well even when domains have equal credits. The BlackScholes run used in this experiment generates 2 million options over 512 iterations in all our domains. Figure 13(a) shows the distribution of throughput values in Million options/sec, as explained earlier. While XC and SLAF see high variation due to higher dependence on driver scheduling and no attempt for CPU and GPU coscheduling, they still perform at least 33% better than None when comparing the medians. AugC and CoSched add an additional 4%–20% improvement as seen from Figure 13(a). The higher performance seen with Dom1 and Dom3 for total work done in Figure 13(b) in case of AugC and CoSched is because of the lower signaling latency seen by the incoming and outgoing domain backend threads, due to their co-location with the scheduling thread and hence, the affected call ordering done by the NVIDIA driver (which is beyond our control).




a) Throughput distribution b) Average work done c) Latency distribution

**Figure 14:** Performance of scheduling schemes [BS]—Credits: Dom1=256, Dom2=512, Dom3=1024, Dom4=256

Beyond the improvements shown above, future deployment scenarios in utility data centers suggest the importance of supporting prioritization of domains. This is seen by experiments in which we modify the credits assigned to a domain, which can further improve performance (see Figure 14). We again use BlackScholes, but with Domain credits as (1) (Dom1,256), (2) (Dom2,512), (3) (Dom3,1024), and (4) (Dom4,256), respectively. The effects of such scheduling are apparent from the fact that, as shown in Figure 14(b), Dom3 succeeds in performing 2.4X or 140% more work when compared with None, with its minimum and maximum throughput values showing 3X to 2X improvement respectively. This is because domains sometimes complete early (e.g., Dom3 completes its designated

runs before Dom1) which then frees up the accelerator for other domains (e.g., Dom1) to complete their work in a mode similar to non-shared operation, resulting in high throughput. The 'work done' metric captures this because average throughput is calculated for the entire application run. Another important point seen from Figure 14(c) is that the latency seen by Dom4 varies more as compared to Dom2 for say AugC because of the temporary unfairness resulting from the difference in credits between the two domains. A final interesting note is that scheduling becomes less important when accelerators are not highly utilized, as evident from other measurements not reported here.

***Coordination respects proportional credit assignments***: The previous experiments use equal amounts of accelerator and CPU credits, but in general, not all guest VMs need equal accelerator vs. general purpose processor resources. We demonstrate the effects of discretionary credit allocations using the BS benchmark, since it is easily configured for variable CPU and GPU execution times, based on the expected number of call and put options and the number of iterations denoted by BS⟨#options,#iterations⟩. Each domain is assigned different GPU and CPU credits denoted by Dom#⟨AccC,XC,SLA proportion⟩. This results in the configuration for this experiment being: Dom1⟨1024,256,0.2⟩ running BS⟨2mi,128⟩, Dom2⟨1024,1024,0.8⟩ running BS⟨2mi,512⟩, Dom3⟨256,1024,0.8⟩ running BS⟨0.8mi,512⟩, and Dom4⟨768,256,0.2⟩ running BS⟨1.6mi,128⟩, where *mi* means million.

Results depicting 'total work done' in Figure 15 demonstrate that coordinated scheduling methods AugC and CoSched deal better with proportional credit assignments. Results show that domains with balanced CPU and GPU credits are more effective in getting work done—Dom2 and Dom3 (helped by high Xen credits)—than others. SLAF shows performance similar to CoSched and AugC due to its use of a feedback loop that tries to attain 80% utilization for Dom2 and Dom3 based on Xen credits. Placement of Dom4 with a high credit domain Dom2 somewhat hurts its performance, but its behavior is in accordance with its Xen credits and SLAF values, and it still sees a performance improvement of at least 18% compared to XC (lowest performance improvement among all scheduling schemes

for the domain) with None. Dom1 benefits from coordination due to earlier completion of Dom3 runs, but is affected by its low CPU credits for the rest of the schemes.

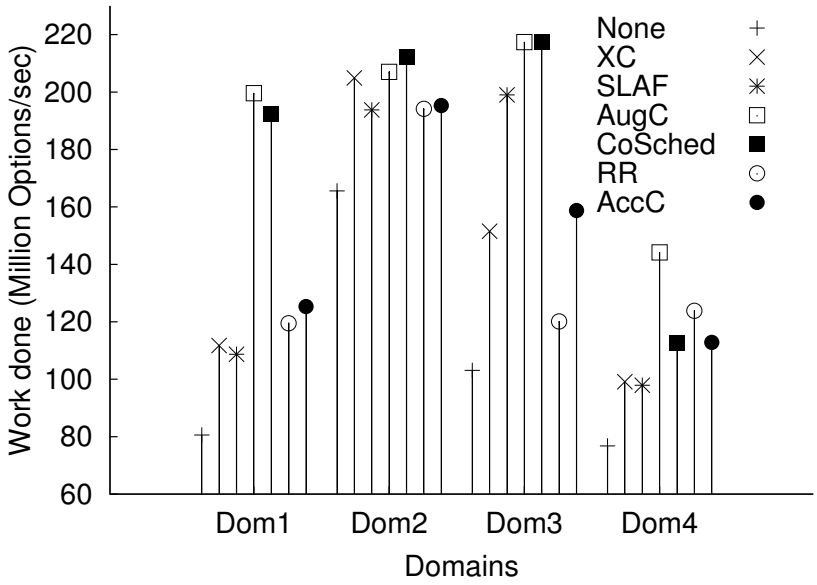
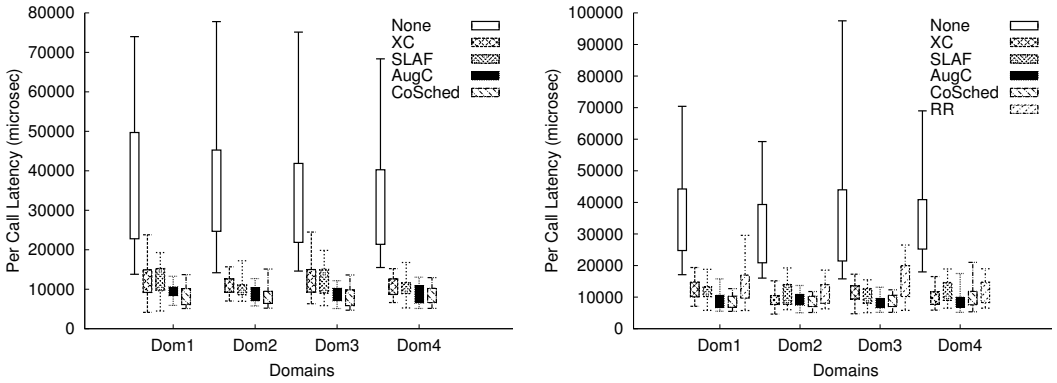



**Figure 15:** Performance of different scheduling schemes [BS] - different Xen and accelerator credits for domains

One lesson from these runs is that the choice of credit assignment should be based on the expected outcome and the amount of work required by the application. How to make suitable choices is a topic for future work, particularly focusing on the runtime changes in application needs and behavior. We also realize that we cannot control the way the driver ultimately schedules requests possibly introducing high system noise and limiting achievable proportionality.

***Coordination is important for latency sensitive codes***: Figure 16 corroborates our earlier statement about the particular need for coordination with latency-intolerant codes. When FWT is run in all domains, first with equal CPU and GPU credits, then with different CPU credits per domain, it is apparent that 'None' (no scheduling) is inappropriate. Specifically, as seen in Figure 16, all scheduling schemes see much lower latencies and latency variations than None. Another interesting point is that the latencies seen for Dom2 and Dom3 are almost equal, despite a big difference in their credit values, for all schemes

except RR (which ignores credits). This is because latencies are reduced until reaching actual virtualization overheads and thereafter, are no longer affected by differences in credits per domain. The other performance effects seen for total time can be attributed to the order in which calls reach the driver.

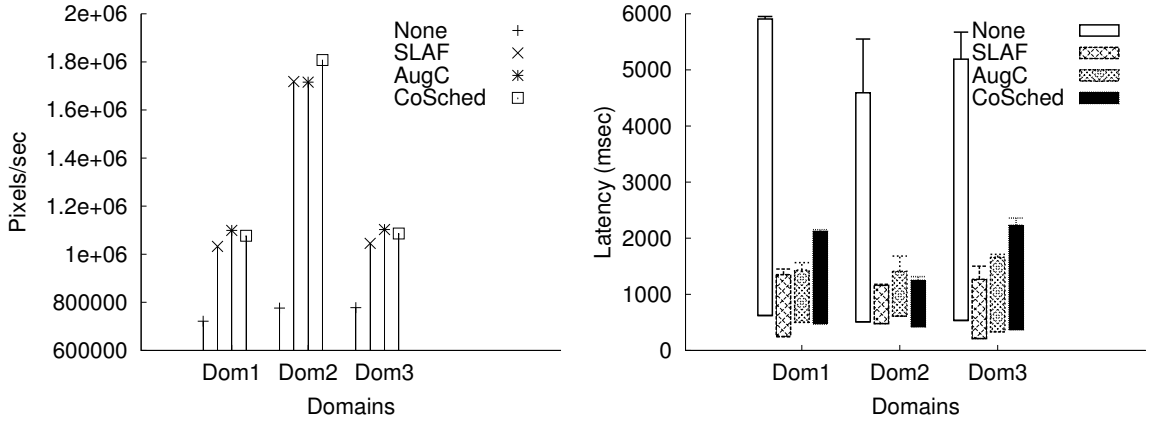



a) All Doms = 256

b) Dom1=256,Dom2=1024, Dom3=2048,Dom4=256

**Figure 16:** Average latencies seen for [FWT]

***Scheduling complex workloads***: When evaluating scheduling policies with the PicSer application, we run three dual-core, 512MB guests on our testbed. One VM (Dom2) is used for priority service and hence given 1024 credits and 1 GPU, while the remaining two are assigned 256 credits, and they share the second GPU. VM2 is latency-sensitive, and all of the VMs care about throughput. Scheduling is important because CPUs are shared by multiple VMs. Figure 17(a) shows the average throughput (Pixels/sec to incorporate different image sizes) seen by each VM with four different policies. We choose AugC and CoSched to highlight the co-scheduling differences. None is to provide a baseline, and SLAF is an enhanced version of all of the credit based schemes. AugC tries to improve the throughput of all VMs, which results in a somewhat lower value for Dom2. CoSched gives priority to Dom2 and can penalize other VMs, as evident from the GPU latencies shown in Figure 17(b). 'No scheduling' does not perform well. More generally, it is clear that coordinated scheduling can be effective in meeting the requirements of multi-VM applications sharing CPU and GPU resources.




a) PicSer Work done b) Latency for GPU processing

**Figure 17:** Performance of selected scheduling schemes for real world benchmark

***Scheduling is not always effective***: There are situations in which scheduling is not effective. We have observed this when a workload is very short lived or when it shows a high degree of variation, as shown in Figure 18. These variations can be attributed to driver processing, with evidence for this attribution being that the same variability is observed in the absence of Pegasus, as seen from the 'NoVirt' bars in the figure. An idea for future work with Pegasus is to explicitly evaluate this via runtime monitoring, to establish and track penalties due to sharing, in order to then adjust scheduling to avoid such penalties whenever possible.

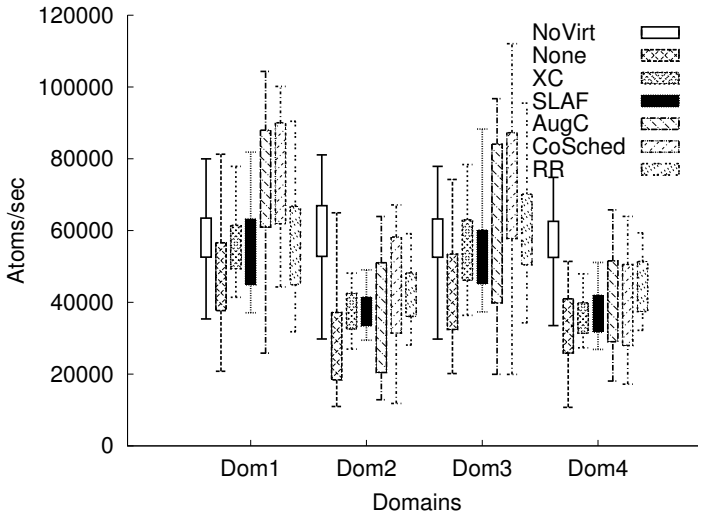



**Figure 18:** [CP] with sharing

***Scheduling does not affect performance in the absence of sharing***: When using two, three, and four domains assigned equal credits, with a mix of different workloads, our measurements show that in general, scheduling works well and exhibits little variation, especially in the absence of accelerator sharing. In all these experiments, Dom1 runs *MC* ⟨ *256, 24mi* ⟩, Dom2 runs *BOp* ⟨ *16384* ⟩, Dom3 runs *PI* ⟨ *2048* ⟩ and Dom4 in the case with four domains runs CP ⟨ 40000 ⟩. Per domain results for the various configurations are shown in Figures 19.a)-i), respectively.

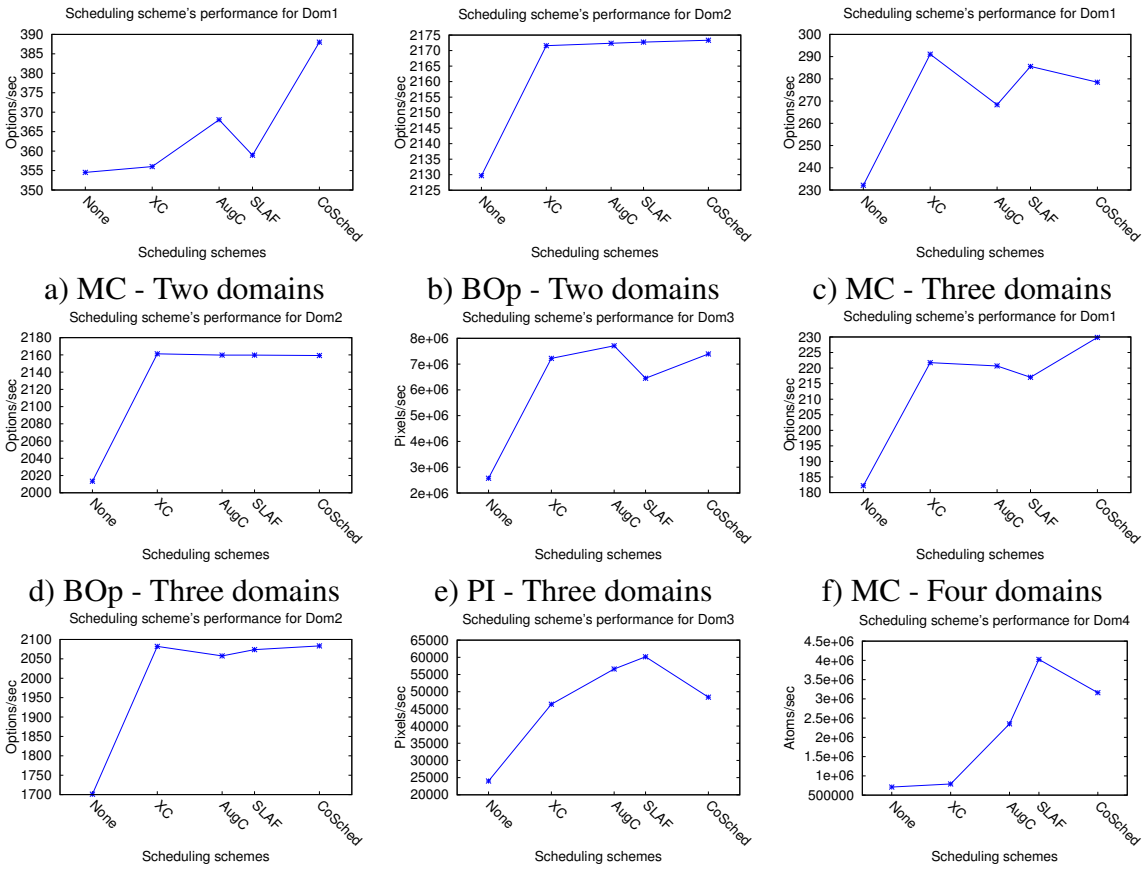



**Figure 19:** Performance of different scheduling schemes for two, three, four domains running different benchmarks and all assigned equal credits in each experiment

For the measurements shown in Figure 19, of particular interest are the results of the experiment with three domains, since it shows both the nonsharing vs. sharing case. In these examples, Dom1 and Dom3 share a CPU as well as a GPU, whereas Dom2 has

its own GPU. The performance of Dom2 is uniformly better for all scheduling schemes whereas Doms 1 and 3 see variations. Another interesting observation that can be made in case of the experiment with four domains. Co-scheduling does not help either Dom3 or Dom4 because they execute far fewer calls and hence are not affected by CPU scheduling as much. They can benefit more from smoothened fair share of the processors.

***Scheduling overheads are low***: We report the worst case scheduling overheads seen per scheduler call in Table 2, for different scheduling policies. MS in the table refers to the Monitor and Sweep thread responsible for monitoring credit value changes for guest VMs and cleaning out state for non-existing VMs. Xen kernel refers to the changes made to the hypervisor CPU scheduling method. Acc0 and Acc1 refer to the schedulers (for timer based schemes like RR, XC, SLAF) in our dual accelerator testbed. Hype refers to the user level thread run for policies like AugC and CoSched for coordinating CPU and GPU activities.

**Table 2:** Backend scheduler overhead

| Policy | MS ($\mu$sec) | Xen Kernel ($\mu$sec) | Acc0/Hype ($\mu$sec) | Acc1 ($\mu$sec) |
|---|---|---|---|---|
| None | 272 | 0.85 | 0 | 0 |
| XC | 1119 | 0.85 | 507 | 496 |
| AugC | 1395 | 0.9 | 3.36 | 0 |
| SLAF | 1101 | 0.95 | 440 | 471 |
| CoSched | 1358 | 0.825 | 2.71 | 0 |

As seen from the table, the Pegasus backend components have low overhead. For example, XC sees $\sim$0.5ms per scheduler call per accelerator, compared to a typical execution time of CUDA applications of between 250ms to 5000ms and with typical scheduling periods of 30ms. The most expensive component, with an overhead of $\sim$1ms, is MS, which runs once every second.

### 4.5.4 Discussion

Experimental results show that Pegasus, built on top of GViM, efficiently virtualizes GPUs and in addition, can effectively schedule their use. Even basic accelerator request scheduling can improve sharing performance, with additional benefits derived from active scheduling coordination schemes. Among these methods, XC can perform quite well, but fails to capitalize on CPU-GPU coordination opportunities for tightly coupled benchmarks. SLAF, when applied to CPU credits, has a smoothing effect on the high variations of XC, because of its feedback loop. For most benchmarks, especially those with a high degree of coupling, AugC and CoSched perform significantly better that other schemes, but require small changes to the hypervisor. More generally, scheduling schemes work well in the absence of over-subscription, helping regulate the flow of calls to the GPU. Regulation also results in lowering the degrees of variability caused by un-coordinated use of the NVIDIA driver.

AugC and CoSched, in particular, constitute an interesting path toward realizing our goal of making accelerators first class citizens, and further improvements to those schemes can be derived from gathering additional information about accelerator resources. There is not, however, a single 'best' scheduling policy. Instead, there is a clear need for diverse policies geared to match different system goals and to account for different application characteristics.

Pegasus scheduling uses global platform knowledge available at hypervisor level, and its implementation benefits from hypervisor-level efficiencies in terms of resource access and control. As a result, it directly addresses enterprise and cloud computing systems in which virtualization is prevalent. Yet, clearly, methods like those in Pegasus can also be realized at OS level, particularly for the high performance domain where hypervisors are not yet in common use. In fact, we are currently constructing a CUDA interposer library for non-virtualized, native guest OSes, which we intend to use to deploy scheduling solutions akin to those realized in Pegasus at large scale on the Keeneland machine.

## *4.6 Related Work*

Prior work on GPU virtualization has used the OpenGL API [57] or 2D-3D graphics virtualization (DirectX, SVGA) [21]. In comparison, Pegasus operates on entire computational kernels more readily co-scheduled with VCPUs running on general purpose CPUs. Besides proposing a high performance virtualization solution, we thoroughly evaluate the approach, develop and explore at length the notion of coordinated scheduling and the scheduling methods we have found suitable for GPGPU use and for latency- vs. throughput-intensive enterprise codes.

While similar in concept, Pegasus differs from coordinated scheduling at the data center level, in that its deterministic methods with predictable behavior are more appropriate at the fine-grained hypervisor level than the loosely-coordinated control-theoretic or statistical techniques used in data center control [54]. Pegasus co-scheduling differs in implementation from traditional gang scheduling [110] in that (1) it operates across multiple scheduling domains, i.e., GPU vs. CPU scheduling, without direct control over how each of those domains schedules its resources, and (2) because it limits the idling of GPUs, by running workloads from other aVCPUs when a currently scheduled VCPU does not have any aVCPUs to run. This is appropriate because Pegasus co-scheduling schemes can afford some skew between CPU and GPU components, since their aim is not to solve the traditional locking issue.

Recent efforts like Qilin [68] and predictive runtime code scheduling [42] both aim to better distribute tasks across CPUs and GPUs. Such work is complementary and could be used combined with the runtime scheduling methods of Pegasus. Upcoming hardware support for accelerator-level contexts, context isolation, and context-switching [76] may help in terms of load balancing opportunities and more importantly, it will help improve accelerator sharing [21].

## *4.7 Conclusions and Future Work*

This research advocates making all of the diverse cores of heterogeneous manycore systems into first class schedulable entities. The Pegasus virtualization-based approach for doing so, is to abstract accelerator interfaces through virtualization and then devise scheduling methods that coordinate accelerator use with that of general purpose host cores. This way, Pegasus enables creation and scheduling of a VM's accelerated or compute personality. The approach is applied to a combined NVIDIA- and x86-based GPGPU multicore prototype, enabling multiple guest VMs to efficiently share heterogeneous platform resources. Evaluations using a large set of representative GPGPU benchmarks and computationally intensive web applications result in insights that include: (1) the need of coordination when sharing accelerator resources, (2) its critical importance for applications that frequently interact across the CPU-GPU boundary, and (3) the need for diverse policies when coordinating the resource management decisions made for general purpose vs. accelerator cores, as indicated by the thesis statement in Chapter 1.

**Thesis discussion:** Pegasus recognizes architectural differences in the computational resources, present within the heterogeneous platform we have discussed, at different layers of systems software stack like the hypervisor—by introducing aVCPU scheduling—and the guest OS—by exposing the availability of GPUs to applications which can use them. The virtualization techniques used to enable accelerator VCPUs makes it possible to apply scheduling logic to a common pool of resources and coordinate scheduling between the CPU and GPU scheduling domains. Distinguishing between the resources, tuning the scheduling policies to work in a favorable manner, and coordinating actions of the different scheduling domains, leads to the improvement in performance and platform utilization despite the architectural differences, as verified by our experimental evaluation.

**Future work:** Certain elements of Pegasus remain under development and/or are subject of future work. Admission control methods can help alleviate certain problems with

accelerator sharing, such as those caused by insufficient accelerator resources (e.g., memory). Runtime load balancing across multiple accelerators would make it easier to deal with cases in which GPU codes do not perform well when sharing accelerator resources. Static profiling and runtime monitoring could help identify such codes. There will be some limitations in load balancing, however, because of the prohibitive costs in moving the large amounts of memory allocated on completely isolated GPU resources. This restricts load migration to cases in which the domain has no or little state on a GPU. As a result, the first steps in our future work will be to provide Pegasus scheduling methods with additional options for accelerator mappings and scheduling, by generalizing our implementation to use both local and non-local accelerators (e.g., when they are connected via high end network links like InfiniBand). Despite these shortcomings, the current implementation of Pegasus not only enables multiple VMs to efficiently share accelerator resources, but also achieves considerable performance gains with its coordination methods.

# CHAPTER V

# MONTAGE: KINSHIP SCHEDULING FOR EFFICIENT EXECUTION OF VIRTUAL MACHINES ON ASYMMETRIC MULTI-CORE PLATFORMS

As mentioned earlier, processor architects are continuing to evolve multicore chips from homogeneous systems (replicas of identical cores), to those that exhibit shared or single ISA heterogeneity [65, 53] on the same die. Contributing to this trend are power delivery constraints that limit the number of full-featured cores on a single chip [72], as well as the proven utility, in terms of performance and power consumption, of specialized cores like those used for accelerating computations [20, 106], network processing, or cryptographic tasks. As indicated by recent announcements by Intel, IBM and AMD of on-chip accelerators [40, 4, 97, 43], processors with cores that differ in their performance and functional properties will increasingly become the basis for future cloud and data center systems.

Efficiently using these platforms poses significant challenges to system software. These can be summarized by asking the following questions: (1) How best to map workloads to different core resources? (2) How to do so for the wide variety of workloads present in cloud and data center systems, which range from computationally expensive codes like parallel simulations and decision engines, to memory-, IO-, and disk-intensive tasks. (3) How to cope with server consolidation, which causes each machine to observe a dynamic mix of applications and application behaviors, with further complications caused by codes that exhibit IO-heavy application phases interspersed with compute-intensive ones [10].

As one example, consider representative workloads from the Parsec suite [12], Hadoop [6] sort [sort-1G] and Hadoop wordcount [wc-240M], and IOzone [114]. Next, consider the heterogeneous research platform depicted in Figure 21.a). This shared ISA machine is

comprised of cores with entirely different performance properties: an Intel Xeon X5450 core and an Atom 330 core (both running at 1.6GHz) able to share aggregate RAM and IO resources. When running workloads on this machine, applications see distinct differences in performance depending on their nature and the core mappings being used. Specifically (Figure 20), the performance of the computationally intensive PARSEC workloads is heavily affected by core type for their native (large) data sets. In contrast the memory/IO-bound

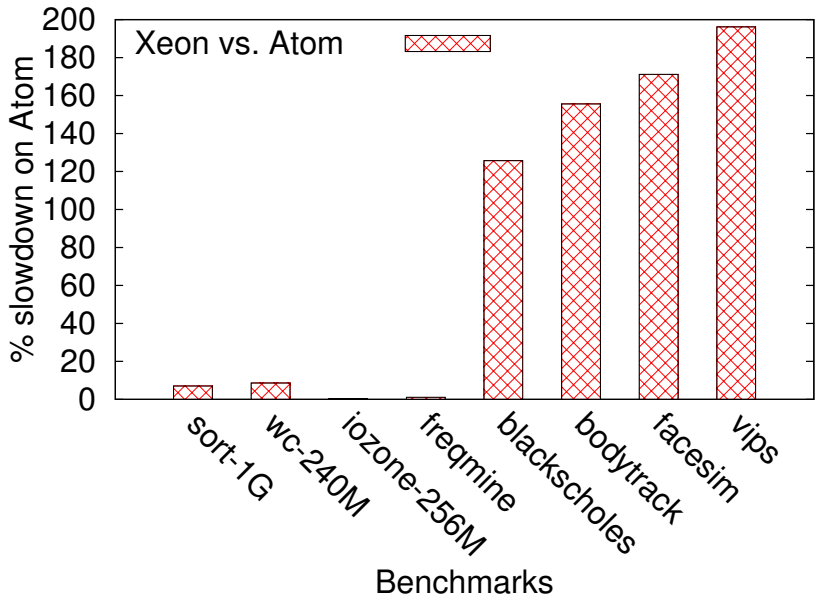
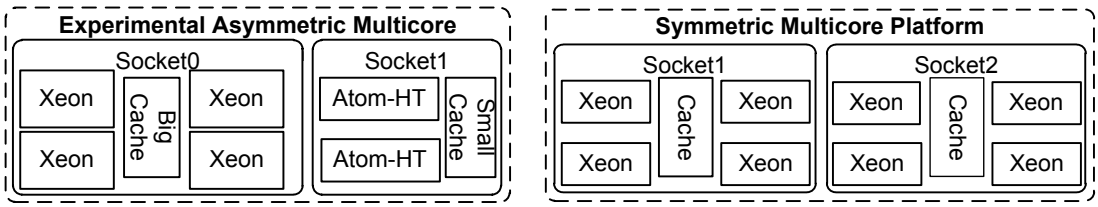



**Figure 20:** Most PARSEC benchmarks show distinct performance benefit on Xeon cores. IOzone, Hadoop sort and wordcount exhibit similar performance on large or small cores.

nature [5] of freqmine, sort, wordcount and IOzone results in only modest differences in performance on the platform's Atom vs. Xeon processors.




a) Intel Xeon-/Atom-based experimental platform

b) Intel Xeon-based traditional platform

**Figure 21:** Sample asymmetric vs. symmetric platforms

This raises further questions: How to make system software aware of application-level sensitivities in the performance experienced on different types of cores? How to best match

core capabilities to application characteristics? Or stated more generally (Figure 22), how to devise scheduling methods that use application workload characterizations to efficiently use heterogeneous core resources.

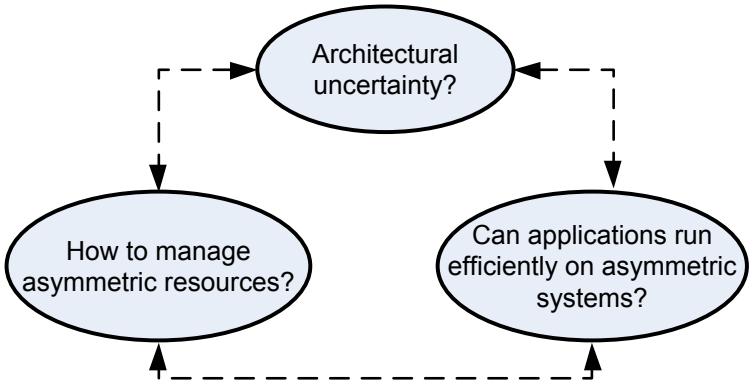



**Figure 22:** Challenges for systems research in asymmetry

The Montage system described in this chapter addresses the challenges and opportunities presented by asymmetric multicore platforms, whose cores differ in their performance or functional properties. For such processors, Montage implements an approach, framework, and methods for creating asymmetry-aware hypervisors. With our system, arbitrary guest virtual machines (VMs) and their applications can efficiently run on asymmetric hardware platforms. Montage, being hypervisor-based, can shield guest OSs and applications from new generations of asymmetric hardware, reducing configuration changes required as data-center infrastructure evolves. This is done by enriching the hypervisor with methods that manage VM execution in ways that are cognizant of workload characteristics in order to best leverage asymmetric platform resources. In particular, Montage offers the novel method of *kinship-based scheduling*, which generalizes prior affinity-based scheduling to also take into account platform asymmetries and workload differences. Through Montage, we make the following technical contributions:

- *Kinship metric*: a new metric for asymmetric systems that extends the traditional notion of affinity used on homogeneous hardware to capture and exploit performance and functional asymmetries in underlying hardware resources. It is defined for all

virtual CPU and physical CPU pairs, for virtual machines to be scheduled on the platform.

- *Kinship-based scheduling:* improves existing hypervisor sche-duling to use kinship values in order to better match the characteristics of application workloads to the capabilities of asymmetric core resources. Characteristics may be stated by hints provided by guest VMs and/or determined by Montage using hardware-level performance counters.

- *Experimental evaluation:* underlines the importance of kinship-based scheduling for realizing the opportunities presented by asymmetric hardware, as well as the potential overheads experienced with its use. We demonstrate asymmetry-awareness for two physically asymmetric platforms, and emulate asymmetry in a third.

Experimental evaluations reveal significant performance improvement for Montage compared to the asymmetry-unaware Xen scheduler. For example, two VMs running benchmarks from PARSEC, Hadoop, and IOzone see an improvement of approximately 6% and 12%, respectively, on the platform shown in Figure 21.a. An emulated functional asymmetry involving cryptographic asymmetry on Xeon platform results in 2X performance improvement without affecting the performance of other apps, (see Section 5.5). Further, improvements occur with small, almost negligible, standard deviation in per-run performance.

The utility and use of hardware asymmetry are ongoing active topics of research. Architectural studies demonstrate power/performance advantages [63, 53], and system support shows value in exploiting performance asymmetries [50, 91, 37]. Other recent work considers power efficiency [34], by asking questions about the value of asymmetries on-chip vs. at larger granularities like different data-center racks. More explicitly, there has been work targeting operating systems and hypervisor schedulers (i.e., the resource management layer in typical cloud systems). Such work has addressed specific types of asymmetries,

as with hypervisor schedulers focused on general purpose processors [27] and potential asymmetries in processor speeds [44] or on NUMA properties [86]. With the exception of [65], none of these efforts focus on the functional asymmetry seen in newer platforms. In comparison, Montage's approach can be used to deal with a broader range of asymmetries present in this rapidly evolving hardware space. In essence, Montage provides a framework for creating *asymmetry-aware* hypervisors that permit cloud and data center systems to adjust to continually and rapidly evolving asymmetric processor hardware. Montage does so (1) by capturing the asymmetric nature of underlying multi-core platforms – with respect to both performance and functional hardware properties, (2) by providing resource management methods that efficiently utilize asymmetric resources for dynamic data center and utility cloud workloads, and (3) by selectively exposing asymmetries to those guest VMs that wish to exploit them. Extensions to Montage can explore additional possibilities for leveraging asymmetry, such as to improve power consumption or meet power caps under consolidation by turning off idle cores or exploiting deep core idle states.

The remainder of this chapter is organized as follows. Section 5.1 outlines the Montage architecture and explains the rationale behind the design and its various components. Section 5.2 describes the kinship model, the major contribution of this dissertation, with its instantiation implemented in Xen described in Section 5.4. Experimental evaluations appear in Section 5.5. Related work is described in Section 5.6, followed by conclusions and future work.

## *5.1 System Architecture*

Montage is a software architecture for hardware platforms that display both performance as well as functional asymmetries. Montage handles demands from multiple guests by making decisions based on both, differences in guest characteristics and asymmetries in hardware. Further, since platforms are still evolving, Montage is designed to address a range of asymmetries. The Montage architecture shown in Figure 23 has the components

described below.

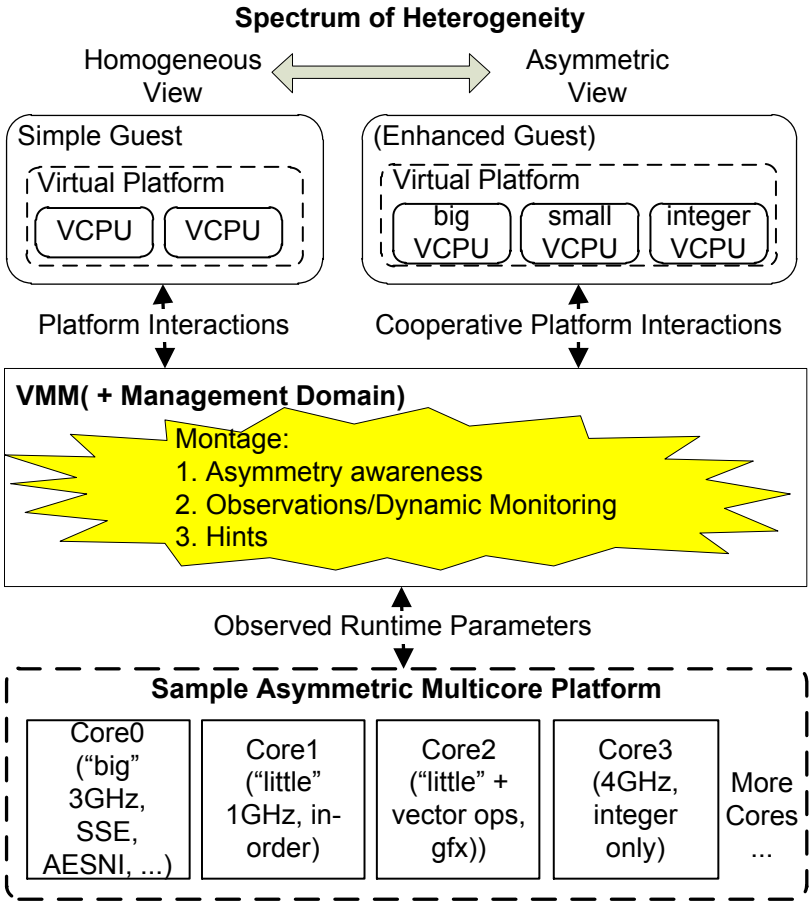



**Figure 23:** Montage Architecture for asymmetric platforms

**Asymmetry-aware hypervisor**: is the core of the Montage framework. Asymmetry awareness involves modifications to recognize asymmetric hardware and to better schedule VCPUs (virtual CPUs), possibly of different personalities on the cores carrying matching 'tags'. Guest operating systems or applications need not be changed. The figure shows the VMM and its management domain, used in hypervisors like Xen or Microsoft's vHype. Montage has been implemented and evaluated with Xen (see Section 5.4).

**Observations or dynamic monitoring:** of VCPU behavior using hardware performance counter data can support a smart hypervisor in deducing the phase of computation. Observation also implies handling instruction faults due to functional asymmetry followed by improving the VCPU-PCPU mappings (see section 5.4). Dynamic monitoring is a complex area of research and sophisticated details are beyond the scope of this research.

**Asymmetry-aware guest OS with aware hypervisor:** is an improvement over the previous step, because the OS also acknowledges asymmetries by scheduling tasks/threads on the right kind of VCPUs, as exposed to it by the hypervisor. Hypervisor awareness is still required in order to properly map VCPUs to PCPUs for all guests in the system. Previous asymmetry research [50, 53, 65, 91] has considered guest-level scheduling, demonstrating the utility of various hardware asymmetries and offering useful insights. Our future work will leverage this by integrating such OS-level functionalities with Montage hypervisor-level support provided by the Montage framework.

**Cooperative platform interactions or hints**: can further improve system performance through a collaboration interface between an aware guest or an administrator and the hypervisor. This is because the hypervisor has global knowledge of the hardware platform and the virtual machines it is managing, while the guest OS is better positioned to understand application behaviors (e.g., phase changes), current computational characteristics, etc. The research problem of determining and then using appropriate collaboration channels is discussed in greater detail in Chapter 6.

While the architecture above is described using a paravirtualized solution, the concepts presented in this chapter can be adopted by other virtualization solutions or even extended to OS scheduling.

## *5.2 Kinship Model for Asymmetric Platforms*

Montage addresses hardware asymmetries for shared or single ISA architectures. Specifically, kinship and kinship-based scheduling consider the broad range of asymmetries shown in Figure 23 and described below:

**Performance asymmetry**: covers variations in processing capabilities due to core frequency scaling, differences in-order vs. out-of-order behavior or in internal instruction issue/commit widths, different LLC (last-level cache) sizes, and so on. Irrespective of the cause of performance asymmetry, the outcome is the difference in performance observed

for application programs running on different cores.

**Functional asymmetry**: given the importance of acceleration and variety in workloads, it is useful to prepare for hardware where some cores on the chip are better customized for certain tasks than others. We assume the existence of a common subset of ISA on all cores (shared-ISA) with some cores enhanced by special instructions to support certain classes of applications. The 'fault and migrate' model permits applications to run on such 'shared ISA' cores [65]. In this model, a VM experiences a fault when it uses an 'unsupported' instruction, which then causes the hypervisor to re-map the VCPU to a PCPU supporting the instruction and then continue its execution.

For Montage, then, the scheduling challenge is to correctly associate VCPUs with PCPUs so as to minimize the risk of faulting. Performance asymmetry further aggravates this problem. The following subsection describes the intuition and the general formulation of our kinship model developed to address above asymmetries. The specific instantiation that we have used with Xen on our experimental platforms is discussed in Section 5.3.

### 5.2.1 The Kinship Model

***Premise:*** *the goal is to optimize, during any given scheduling period, the VCPU-PCPU mappings across all VCPUs running on the asymmetric hardware with respect to their current runtime characteristics, subject to constraints imposed by VM credits or priorities, or in short, to match VM personalities to platform components with matching tags.*

Therefore, the **kinship metric** can be defined as "a numerical value computed based on workload and physical hardware characteristics, in order to find the best match of VCPUs to PCPUs (not necessarily the fastest), to enable efficient execution and effective utilization". Thus, the kinship computed for every VCPU and PCPU pair in our system is used to determine suitable VCPU-PCPU mappings within the scheduler. VM credits are used as tie breakers and to decide the actual running times of VCPUs on their target PCPUs. The

kinship-based scheduling algorithm runs periodically to update these mappings and account for differences in workload characteristics, including those caused by phase changes and/or in platform capabilities. The actual runtime characteristics accounted for depend on (1) the significance and measurability of their effects on VCPU execution, and (2) the possible sources of asymmetry present on the platform.

To better explain the kinship metric and kinship-based scheduling, we formulate use-cases with: (1) benchmarks from the PARSEC suite, (2) hadoop sort, (3) iozone, and (4) an Intel-published encryption benchmark (we call it AES-bench) used to test the performance of the new AES [28] instruction added to Intel's Xeon-family processors.

***Use-case1* for performance asymmetry**: consider scheduling two virtual machines, VM1 running *ferret*, *iozone* for a file size of 512MB in automatic mode, and VM2 running *streamcluster*, *freqmine*, *hadoop-sort* of GB-size data and another similar *iozone* instance. The target for this use-case is the dual-socket platform shown in Figure 21.a) with one Xeon and one Atom socket. Of these benchmarks, iozone, freqmine and sorting are mostly unaffected by the speed and cache size in the Atom processor as shown in Figure 20 in the introduction.

***Use-case2* for functional asymmetry:** while compute-intensive VCPUs typically perform better on fast cores, scheduling a VCPU on a slower core may be advantageous if that core offers some special instruction that can heavily speed up the VCPU. Consider two VMs, VMx intending to run *AES-bench* with a large data set and *dedup* on its two VCPUs, and VMy intending to run *AES-bench* with a small data set and *swaptions* on its two VCPUs. Of these benchmarks, *dedup* and *AES-bench-small* are marginally affected by speed, while *swaptions* and *AES-bench-large* suffer significant slowdown on slow cores. AES-bench, of course, runs faster with AESNI instructions; it experiences great slowdown if AES is emulated in software. We evaluate this use case on a different asymmetric platform comprised of four Xeon cores across two sockets (2 cores from Socket1 and two from Socket2). The AESNI instruction is disabled in Socket1, which forces software emulation.

One out of the two cores from each socket is slowed down to run only at 50% efficiency. All four cores are shared by the two different VMs.

With these use cases, we can now define the kinship model as well as explain kinship-based scheduling. The *kinship value* ascribed to a VCPU,PCPU pair separates performance from functional hardware asymmetries.

***Performance component of kinship*** - $E_p^v$ defines the VCPU-PCPU kinship governed by the presence of performance asymmetry. It is defined as follows:

$E_p^v$ = *g(observed behavior, VM credits, given cpu / mem expectation, current CPU / memory load)*

The first parameter accounts for the observed behavior of a VCPU $v$ on some PCPU $p$ which for example, could vary at runtime, e.g., across different phases of the workload scheduled on this VCPU. The VM credits represent the priority or credit assignments made for a particular VM by the system administrator. An administrator can also specify compute, cache, memory and/or IO expectations, depending on the VCPU workload. For example, it is a means to specify whether the workload to be run on a VCPU would be say, cache intensive vs. cpu intensive. Expectations are currently stated as hints about workload behavior, e.g., MOSTLY_CPU_INTENSIVE or MOSTLY_CACHE_INTENSIVE, rather than requiring detailed specifications such as "70% of a CPU clocked at 2GHz". Their provision by developers or users is known to be straightforward for high performance applications that are extensively and repeatedly tuned. It may be difficult to determine them for general server applications, which is why (1) they are optional hints and (2) they can be determined online with automated methods, described in more detail in Section 5.4.

Given these parameters for $E_p^v$, initially, there is no observed behavior for Use-case1 and hence, $E_p^v$ is not affected by it for any of the VCPUs. Assuming equal credits for both VMs, the difference will be visible among the VCPUs with regard to their CPU/memory expectations. VCPUs running ferret and streamcluster could have hints labeling them as MOSTLY_CPU_INTENSIVE. This will lead to a higher value for kinship between those

VCPUs and the PCPUs with better computational capabilities (i.e., the Xeon cores in this case). The rest of the VCPUs running freqmine, iozone, and hadoop-sort will get a chance to use the Xeons only when those cores are not currently used. Else, these workloads will be scheduled on the Atoms. The outcome is a system in which VMs not only receive computational cycles according to the credits assigned to them, but in addition, their VCPUs are run in ways that improve the performance of compute intensive VMs, without compromising the performance experienced by others. We note that this scheme differs from the previous hypervisor-level solutions for asymmetry [44] which instead, give a fair-share of the fast cores to all VCPUs in proportion of their credits rather than behavior, since such a scheme will lead to an overall degradation in possible system performance.

***Functional component of kinship*** - $F_p^v$ defines the VCPU-PCPU kinship governed by the presence of functional asymmetry. This factor requires observation at the hypervisor level, but can also be assisted by collaboration between the guest and VMM scheduler.

$F_p^v$ = *h(observed behavior, given execution category, cpu capabilities)*

The first parameter is an observed behavior that refers to the functioning of a VCPU $v$ on PCPU $p$ over a moving time window. For example, if a VCPU $v0$ faulted due to a missing instruction on PCPU $p2$, the kinship between $v0$ and $p2$ is affected negatively, thus making it less likely for $v0$ to be scheduled on $p2$ in the near future. The moving time window is used to make it possible for $v0$ to return to $p2$, if required by say, performance constraints, when no faults are seen in a more current time frame. The 'execution category' could be set by an administrator or an aware guest, if workload behavior is known. Categories of execution ($CAT^v$) can be defined both for VCPUs, depending on the expected workload, and for PCPUs, depending on the instructions supported. For example, a VCPU that is going to run a vector application and extensively use Intel's SSE extensions, can be marked as such. CPU capabilities are calibrated to indicate whether it is a general purpose CPU, has some special vector capabilities, and so on.

Considering Use-case2, the two VCPUs running AES-bench will belong to the same

category, say $CRYPTO$. The Xeon cores being the targets in this scenario will be marked as $CRYPTO$ capable if they support the AESNI instructions. Hence, compared to the other VCPUs, the CRYPTO VCPUs will have a higher functional kinship value for the corresponding CRYPTO PCPUs.

***Kinship*** - expresses how well a certain VCPU matches the capabilities of a PCPU. It is denoted as $K_p^v$ in our equations, and it associates a VCPU $v$ with a PCPU $p$. Distinguishing it from the notion of affinity formulated for identical cores, it has both the performance/efficiency and functional kinship components described above. Kinship components incorporate observed system parameters as well as the input a domain might provide regarding the CPUs it expects, their characteristics, etc. For guests that are not prepared for asymmetry, suitable values are assumed in the equations we define. Kinship can therefore, be represented as: $K_p^v = f(E_p^v, F_p^v)$

Revisiting Use-case2, as discussed earlier concerning the functional kinship component, the VCPUs running AES workload have higher kinship for the PCPUs supporting AESNI. Kinship-based scheduling, however, considers both functional and performance asymmetries. As a result, since the particular platform under consideration also has speed asymmetry, the performance component of kinship will result in the VCPU running the large AES data set to have the highest kinship with the fast core supporting AES, whereas the smaller AES will be scheduled on a core with AES support but only half the performance. Further, since swaptions is more compute intensive than dedup, it will have higher kinship with the faster core. The results of this execution are shown in Section 5.5, in which we show that kinship-based scheduling is much superior to the asymmetry-unaware, standard Xen scheduling techniques.

In summary, kinship components are formulated so that they can make use of (1) user provided hints and/or dynamic runtime observations, and (2) knowledge about platform configuration, resulting in the composite value $K_p^v$, used for asymmetric scheduling. Figure 24 summarizes the formulation. To address the kinds of asymmetries described above,

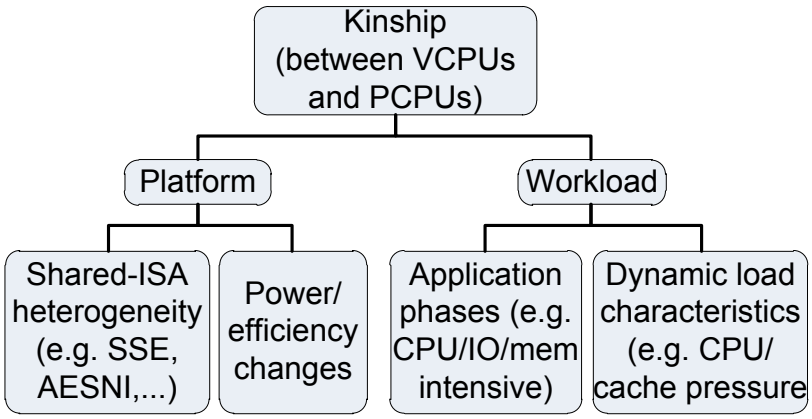



**Figure 24:** Kinship values are calculated per VCPU-PCPU pair to help schedule on asymmetric platforms

*kinship-based scheduling* makes decisions based on *kinship values* between the VCPUs of guest domains and the PCPUs present in the system. The parameters used to calculate kinship values are either calibrated for the system and supplied by an external source, or they are populated by the system at runtime. When there is no user input, the model starts with values that represent the most common behavior; they are updated as VM execution progresses. This requires active monitoring by the hypervisor/management domain. The detailed computation of kinship used in our implementation is described in Section 5.3.

## *5.3 Kinship Calculation*

This section presents the equations used to compute the different components of kinship used in our implementation. They are shown to work with the experiment scenarios in Section 5.5. As mentioned in Section 5.2, the quantities used to define the functions are determined by how measurable they are and if they have a quantifiable effect on execution of a VCPU. Adopting a top-down approach for presenting this specific instantiation of the kinship model, Equation (1) shows the primary equation that evaluates kinship values per VCPU-PCPU pair.

$$K_p^v = E_p^v + F_p^v + C_p^v \tag{1}$$

where $E_p^v$ and $F_p^v$ are the performance and functional components of kinship $K_p^v$ as mentioned earlier in Section 5.2.1 and $C_p^v$ indicates whether VCPU $v$ is allowed to run on PCPU

$p$. We classify the asymmetries based on performance vs. functional due to differences in the way these asymmetries manifest themselves and the effect they have on the workloads. For instance, a functional asymmetry like the absence of some group of instructions would cause a VCPU to fault and disrupt its execution, whereas a frequency difference between cores would simply result in different execution times witnessed by the workload on the different PCPUs. In response, the kinship model should be able to incorporate such disparities efficiently. An additional important factor, especially in future large scale many-core systems, is the ability to limit the group of PCPUs on which a VCPU can be scheduled. This could be done for various reasons like memory distances, non-uniform cache effects, latency of communication between various resources as more and more cores get added and so on. $C^v$ denotes such a permissible set of PCPUs. All the quantities discussed here are defined per VCPU-PCPU pair. Therefore, unless noted otherwise, we will omit the $v$ and $p$ indexing in the following equations. Since the VCPUs with highest kinship values get scheduled first, so as to allocate them to their best matched PCPUs, the kinship value $K_p^v$ can be multiplied by a *fairness token* which is a token passed around different VMs, in a round robin order, to boost the holder VM's priority for fast core access. This can help us distribute fast core cycles among all VMs, similar to the ideas described in related work like [44].

Now, each component in Equation 1, at any time $t$ is defined as:

$$E = w_E * E_t \tag{2a}$$

$$F = w_F * F_t \tag{2b}$$

$$C = \begin{cases} 0 & \text{if } p \in cpupool^v, \\ -\infty & \text{otherwise} \end{cases} \tag{2c}$$

or we can express the overall kinship value as a dot product of the weight vector with the

vector composed of individual kinship components as shown in Equation 3

$$K_p^v = \begin{bmatrix} w_E & w_F & 1 \end{bmatrix} \cdot \begin{bmatrix} E_t \\ F_t \\ C_t \end{bmatrix} \tag{3}$$

The weights $w_E$ and $w_F$ control the effect of each performance and functional asymmetry component on the overall kinship value. We set them to equal values (i.e., 1) in the experiments presented in Section 5.5. We will now discuss the individual elements of the kinship component vector shown in Equation 3.

### 5.3.1 Performance Kinship

Performance kinship is defined to capture elements in the system that can affect the performance achieved by the workloads and the overall system. We have described various physical and software related effects that can cause performance asymmetry in a platform in the machine model discussed in Chapter 2. We now show how our kinship implementation takes into account these various elements of performance asymmetry.

Before we define the equation for $E_t$, we need to express the resources that cause various asymmetries and the role they play in determining the scheduling of the various VCPUs on the corresponding PCPUs. Let $R$ be the set of all resources in the system. Let $R_i$, $R_j$ be subsets of $R$. Some $R_i$ and $R_j$ may have shared resources e.g. $\{cache, memory, IO, \ldots\}$ or they may have some exclusive resources like $\{cpu\}$. And $R_x \cap R_y \neq \emptyset$, in general. This set, as alluded earlier, is determined by measurable elements of the system. Since the objective behind our work is to find near-optimal mappings of VCPUs to PCPUs, our definition of kinship is inclined as such. So we associate all these resources with individual PCPUs in the system. In other words, $R_x$ in our discussion is the set of resources associated with some PCPU say $p$. As said earlier, different PCPUs can have common resources e.g. PCPUs in one socket share a cache or in current systems, all PCPUs share the memory and IO subsystems. For simplicity of explanation, lets say we consider the following set $R_x$ of

resources per PCPU $\{cpu_x, cache_x, mem_x, io_x\}$ where $cpu$ is the actual processing core, *cache* is the last level cache in the socket, *mem* is the RAM accessible from the PCPU and *io* denotes the IO subsystem. It is of course possible to define as many elements in this set depending on the degrees of asymmetry, like L1 and individual processing units.

Since kinship is expected to incorporate user specified information, observed workload behavior and load on the resource under consideration at any given time, we define the following matrices of parameters per resource type being accounted for, in the system. Let $r$ denote a resource in our resource set $R$ and now on we will typically use it to denote some resource belonging to a particular PCPU's resource set $r \in R_x$.

$$I^v = \begin{bmatrix} i_{cpu} & 0 & 0 & 0 \\ 0 & i_{cache} & 0 & 0 \\ 0 & 0 & i_{mem} & 0 \\ 0 & 0 & 0 & i_{io} \end{bmatrix} \quad \ldots \text{for all resources} \tag{4a}$$

$$L = \begin{bmatrix} l_{cpu} & 0 & 0 & 0 \\ 0 & l_{cache} & 0 & 0 \\ 0 & 0 & l_{mem} & 0 \\ 0 & 0 & 0 & l_{io} \end{bmatrix} \quad \ldots \text{for all resources} \tag{4b}$$

$$G^v = \begin{bmatrix} g_{cpu} \\ g_{cache} \\ g_{mem} \\ g_{io} \end{bmatrix} \quad \ldots \text{for all resources} \tag{4c}$$

$$W = \begin{bmatrix} w_{cpu} & w_{cache} & w_{mem} & w_{io} \end{bmatrix} \quad \ldots \text{relative importance of resources} \tag{4d}$$

In the above equations, $I^v$ is a matrix representing the intensity at which each $v$ uses each $r$; $L$ is a matrix representing the load currently experienced by $r$ due to all other VCPUs scheduled to use it; $G^v$ is a vector of values computed from given hints about $v$'s usage of

$r$ normalized to the minimum capability of all resources of the same type in the system. Since any VCPU $v$'s use of a resource $r_m$ associated with PCPU $p$ can possibly have an effect on it's use of another resource $r_n$ associated with $p$, we use matrices to represent both $I$ and $L$. So each row in those matrices contains values reflecting the effect of one resource's observed behavior on all others. However, in our calculations, we assume no such cross effects. For example, when a CPU intensive VCPU runs on some PCPU, we observe performance counters and conclude that the VCPU is CPU intensive shown by the value $i_{cpu}$. However, that observation is independent of the observation that will tell us about it's cache behavior. $G^v$ is simply a vector instead of a matrix because a user specifies per resource expectations or hints for each VCPU for now (future work will look at how user could specify other hints like coordinated execution).

Now, for a given $\{p, v\}$ pair, we define the performance kinship component, at any time t, as:

$$E_t = W.I^v.L.G^v \tag{5a}$$

...which computes to

$$E_t = \sum^{r \in R_x} w_r * I_r * L_r * G_r$$

...where $R_x$for p $= \{cpu, cache, mem, io\}$ in our equations

...OR can be expressed as

$$E_t = w_{cpu} * E_{cpu} + w_{cache} * E_{cache} + w_{mem} * E_{mem} + w_{io} * E_{io} \tag{5b}$$

The individual elements from Equation (5) are calculated as shown in Equations (6),

(7), (8) and (9) and explained in the remaining part of this section.

$$E_{cpu} = I_{cpu} * L_{cpu} * G_{cpu} \quad \ldots \text{For CPU}$$

$$I_{cpu} = \text{observed cpuness from performance counters}$$

$$L_{cpu} = \begin{cases} 1 & \text{at no load,} \\ \frac{1+IOload_p}{CPUload_p} & \text{otherwise} \end{cases}$$

$$IOload_p = \sum^{allvcpusonp} \frac{I_{io}^v * CC^v}{CC_p^{min}} \tag{6}$$

$$CPUload_p = \sum^{allvcpusonp} \frac{I_{cpu}^v * CC^v * CS^v}{CC_p^{min}}$$

$$G_{cpu} = \frac{S_p * EXP_{cpu}^v}{S_{min}} * S_p$$

$$S_p = \text{Calibrated speed for } p$$

$$S_{min} = \text{Slowest speed } \forall p$$

In Equation (6), $G_{cpu}$ is the scaled expectation as calculated above by using the given expectation for a VCPU, the speed ($S$) of the PCPU $p$ under consideration, and the minimum across all PCPUs in the system. The $S_p$ value for all PCPUs is calibrated at runtime using a calibration code since the hypervisor itself has no way of recognizing the kind of speed differences we are dealing with (micro-architectural, software-calibrated and so on, which do not show up in a call to CPUID). The calibration code times a loop executing different operations like addition, subtraction, division and multiplication on a set of variables, taking care to avoid compiler optimizations (necessary to get as accurate a calibration as possible). In the above equations $L_{cpu}$ is the *strength coefficient* or *load factor* for the PCPU. Its computation iterates over all VCPUs attached to the PCPU at the given time and accounts for the load they have introduced on this PCPU, in order to give the remaining strength or capacity available for the new VCPU being considered.

If a VM expects a certain level of performance conveyed by some SLA, the hypervisor's scheduler cannot ignore these differences across cores. A guest OS is assigned credits to specify the proportion of CPU time it receives compared to other guests. So, we define

$CC^v = \frac{S_{min} * credits^v}{S_p}$, called VCPU $v$'s current credit value, and it scales the credits available to a VCPU based on the capability of the PCPU $p$. For example, on a PCPU $p2$ with a speed of 3GHz and a minimum PCPU speed in the system of 2GHz, a VCPU with credits of say, 120 will receive a $CC$ value of 80 on $p2$. This accounts for the differences in speeds of the various PCPUs and leads to fairness [62] in PCPU cycles assigned to the VCPUs in the system. Thus, the $EXP$ values, credit values are all defined with the slowest CPUs or smallest cache sizes as the base. The $CS^v$ value used in calculating $CPUload_p$ incorporates the current runstate of a VCPU $v$ and is defined as $EXP^v_{cpu} * runstate^v$, where $runstate^v$ is 0 if $v$ is blocked (which implies that $v$ is not using cycles on $p$) or 1 if running (which implies that $v$ is using $p$ cycles proportional to its expectations).

$$
\begin{aligned}
&E_{cache} = I_{cache} * L_{cache} * G_{cache} && \ldots \text{For cache} \\
&I_{cache} = \text{observed cacheness from performance counters} \\
&L_{cache} = \begin{cases} 1 & \text{at no load,} \\ \frac{1}{CACHEload_p} & \text{otherwise} \end{cases} \\
&CACHEload_p = \sum^{allvcpusonp} working\_set^v * EXP^v_{cache} \\
&G_{cache} = \frac{l_p * EXP^v_{cache}}{l_{min}} * l_p \\
&l_p = \text{Calibrated cache for } p \\
&l_{min} = \text{Smallest cache } \forall p
\end{aligned}
\tag{7}
$$

Similar to $r = cpu$ from Equation (6), in Equation (7), $G_{cache}$ is the scaled expectation as calculated above by using the given expectation for a VCPU, the cache size ($l$) of the PCPU $p$ under consideration, and the minimum across all PCPUs in the system. The $l_p$ values for all PCPUs is acquired from the Xen calibration code (differences in cache sizes do get noticed in the existing hypervisor code). The runstate considered for cpu does not affect the cache footprint in an obvious way and hence, the $CACHEload$ values just account for the expected cache usage expectation, if specified, and the observed cache usage intensity [48,

49].

$$
\begin{aligned}
E_{mem} &= I_{mem} * L_{mem} * G_{mem} \qquad \ldots \text{For mem subsystem} \\
L_{mem} &= \sum^{allvcpusonp} EXP_{mem} \\
G_{mem} &= EXP_{mem}
\end{aligned}
\tag{8}
$$

$$
\begin{aligned}
E_{io} &= I_{io} * L_{io} * G_{io} \qquad \ldots \text{For the IO subsystem} \\
L_{io} &= \sum^{allvcpusonp} EXP_{io} \\
G_{io} &= EXP_{io}
\end{aligned}
\tag{9}
$$

Equations (8) and (9) are very simplified because quantifying the different ways in which memory and IO affect workloads is beyond the scope of our current work. Therefore, we rely on the given hints from the user, as of now, to determine the behavior of the VCPU showing memory or IO intensity. However, we do characterize a VCPU as IO intensive if it shows large value of RESOURCE_STALLS and characterize it as memory intensive if it shows a large value of LLC_MISSES which are the two counters available on all platforms we have considered.

In general, as seen from the above equations, the $L_r$ values capture the current load on a resource. During the course of our evaluation, we realized that it was important to account for load introduced by VCPUs along all resources while calculating the kinship value of a new VCPU to the PCPU being considered. Therefore, we have now enhanced all the $E_r$ computations, shown in Equations (6), (7), (8) and (9) to use $L_{overall}$ which is computed as $L^v_{overall} = \sum^{allvcpusonp} \left\{ \sum^{r \in R_x} L_r \right\}$. As seen from these equations, the $L$ values are defined to get mathematically smaller as load increases (e.g. inversely proportional to cpuness/memness of VCPUs assigned to PCPU $p$), reducing the overall kinship values in Equation (5), as desired. For the evaluation presented in Section 5.5, we provide the $EXP$ values for all VCPUs to indicate their resource expectations.

### 5.3.2 Functional Kinship

With parameters from Equation (2a) defined, we now discuss the definition of $F_t$ from equation (2b).

$$F_t = MF * FV \tag{10}$$

$MF$ in equation (10) denotes the match factor between a VCPU $v$'s expected category and the PCPU $p$'s supported categories. The VCPU categories were discussed earlier in Section 5.2.1. Categories are assigned to PCPUs when their characteristics are being calibrated. For example, if there are 4 out of say, 8 CPUs in the platform that support SSE extensions while also supporting all other general purpose instructions, those four can be marked $VECTOR$ along with *GENERAL*. So, different PCPUs can coexist in different categories or belong to completely disjoint categories like a GPU (which would purely be $VECTOR$) and x86 CPU (which would be *GENERAL*). This will allow us to accommodate disjoint ISA CPUs in the future. In our implementation, these categories are defined as bitmaps that can be *AND*ed for quick calculation. Hence, $match = CAT^v \& CAT_p$, as used in Equation (11a) for calculating $MF$.

$$MF = \begin{cases} 1 & \text{if } match = 0, \\ match & \text{otherwise} \end{cases} \tag{11a}$$

$$FV = \begin{cases} 1 & \text{if } faults^{TW} = 0, \\ faults^{TW} * \left(-1 + \frac{emulated}{1+emucost}\right) & \text{otherwise} \end{cases} \tag{11b}$$

$FV$ from Equation (10) is the fault value as calculated in Equation (11b). We add up the observed number of faults when a VCPU faults on a particular PCPU over moving time window $TW$. It is possible for a fault to lead to an emulation code which would be slower but won't halt the VCPU execution or cause migration. That is taken care of in the second term of Equation (11b). The variable $emulated$ is a boolean value equal to either 0 or 1 depending on presence or absence of emulation. If $emulated$ is 1, the negative

effect of faults is reduced by its presence in inverse proportion to the cost of emulation (or the slowdown observed). As shown in [65], it can be cost-prohibitive to identify the instruction that causes a fault and hence, the number of faults are maintained independent of the category of instruction causing the fault. For example, if a VCPU first faults due to missing SSE and then due to a missing AESNI [28], it would increment the same fault counter in that time window despite SSE belonging to the $VECTOR$ category and AESNI belonging to $CRYPTO$.

## 5.4 System Implementation

This section explains how the Xen credit scheduler [14] is enhanced with an implementation of the kinship model.

### 5.4.1 Scheduler Modifications

The Xen scheduler schedules VCPUs from different guest VMs in proportion to their credits in a work conserving manner. While trying to maintain the work conserving property of the scheduler, VCPU mappings in Montage are guided by the kinship model. For example, if PCPU $p2$ has an empty ready queue, whereas $p1$ has two VCPUs scheduled on it, the Xen scheduler will try to migrate one of the VCPUs to PCPU $p2$. However, if the kinship values between the VCPUs and $p1$ are higher compared to their kinship values with $p2$, the Montage scheduler will prevent the migration. The idleness of $p2$ is addressed in the next round of accounting, where re-balancing ensures a more even distribution of loads. The algorithms used for Montage scheduling are described next.

#### *5.4.1.1 Algorithms*

For simplicity of algorithm description, consider that the kinship values between VCPUs and PCPUs are maintained in a matrix called the kinship matrix $KM$. Now $KM$ can be used to make decisions about which VCPU to PCPU assignments. An issue for this approach is that the core strength/load ($L_p$) values change whenever some observed value is changed,

which in turn leads to an update of the kinship matrix. Further, this can happen while some VCPU-PCPU mapping is in current use. We avoid this problem by not including $L$ values in calculating these data structures. Instead, those values are included at the time the VCPU-PCPU mapping is being calculated. This increases the number of checks required while making an assignment but keeps the data structures relatively stable once they are calculated using $L = 1$.

Kinship algorithms have been implemented using highly optimized data structures and math operations to reduce the overhead introduced by the incorporation of kinship equations. We present an overhead evaluation of the scheduler in Section 5.5. Further, in order to simplify the explanation of these algorithms, we use a notation in which the ready queue of VCPUs at each PCPU is denoted as $p.readyQ$, the list of PCPUs ordered based on kinship $K_p^v$ per VCPU $v$ as $CPUQ^v$, and the entire set of VCPUs in the system from all of the guest VMs as the list $GL^V$.

**Assignment of a VCPU to a PCPU** – this main scheduling component is described in Algorithm 3. The algorithm uses the value of kinship consisting of the core strength (load factor) $L$, in order to pick the correct PCPU at any given point in time. In the actual implementation, VCPUs are migrated from the currently assigned PCPU to the new one, and corresponding updates are made to the relevant data structures.

**Algorithm 3**: Assigning VCPU to a PCPU at any given time

```
Input: VCPU v and kinship matrix (KM)
Output: Assignment VCPU-PCPU
foreach PCPU p from CPUQ^v do
    kinship1 = K_p^v with actual L;
    kinship2 = K_{p+1}^v with actual L;
    if kinship1 ≥ kinship2 or p last in CPUQ^v then
        Add v to P.readyQ with scaled VM credits for VMM scheduling;
        Update all L;
        break;
```

**Admitting a VCPU in the system** – in order to start a VCPU on as well-matched a PCPU as possible (this is highly beneficial where large working set sizes are concerned),

any hints that might have been offered are incorporated based on the considerations shown in Algorithm 4. So for a new VCPU, kinship values of v with all CPUs in the system are computed and added to $KM$. Then the algorithm in Algorithm 3 takes over to schedule the VCPU on best available PCPU.

**Algorithm 4**: Admitting a VCPU in the system

**Input**: New VCPU v and kinship matrix (KM)
**Output**: Assignment VCPU-PCPU
Calculate $K_p^v \forall p \in PCPUs$ with $L = 1$;
Add $v$ to $GL^V$;
Run Algorithm 3;

**Accounting and re-scheduling**: system configurations change over time, e.g., new VCPUs are added, old ones removed, priorities change, etc. To maintain updated VCPU-PCPU mappings, the algorithm shown in Listing 5 is run in every accounting period.

**Algorithm 5**: Rematching VCPU-PCPU Mappings every accounting period

**Input**: $GL^V$, $p.readyQ \forall PCPUs$, and $KM$
**Output**: VCPU-PCPU mappings $\forall VCPUs$
**if** *Time for remap* **then**
    Assume no load for all PCPUs, maintain other observed values;
    Handle pending VCPU migration from system runs; Reevaluate $KM$;
    **foreach** *VCPU v from list* $GL^V$ **do**
        Run Algorithm 3

The accounting period is chosen on an empirical basis such that the data structures do not change too often while still maintaining sensible mappings. The current Montage accounting period is four times the credit accounting period used by the Xen scheduler. This update can be triggered sooner if frequent changes are made to VCPU or PCPU (like software frequency scaling) properties.

**Observe and modify**: an initial version of the observation algorithm monitors certain performance counters. Algorithm 6 lists the values that trigger changes to the scheduling data structures.

Additional modifications are necessary to make the Xen scheduler asymmetry-aware.

**Algorithm 6**: Monitoring and corresponding modifications

**Input**: VCPU data structures
**foreach** *PCPU P* **do**
    After every VCPU $v$ execution cycle Update working set size for $v$ and $P$;
    Update cpu, cache, mem and io intensity for $v$;
**foreach** *VCPU v fault on PCPU* **do**
    **if** *fault due to functional asymmetry* **then**
        Update $faults^{TW}$;
        **if** *emulation and* $faults^{TW} < threshold$ **then**
            Handle fault in VMM;
            Update $F_T$;
        **else**
            Trigger VCPU migration;

For example, with different generation cores in two sockets of a platform, the VMCS structures [60] have different formats, but the current Xen code maintains this information as globals assuming identical formats. Further, we have implemented calibration code to calculate CPU speed differences, cache variations across sockets, and memory sizes accessible from each core. Calibration also checks for special instructions like the vector SSE and AESNI support, in order to categorize PCPUs correctly for kinship equations.

### 5.4.2 Implementation of the Hints Channel

We have implemented the following interfaces to assist users/administrators in specifying the expected behavior and categories of the VCPUs belonging to some domain:

```
set_vcpu_category(Domain d, Vcpu v, Category c)
set_vcpu_expectation(Domain d, Vcpu v, Expectation e)
```

These have been implemented in our current prototype as hypercalls. The category values belong to a set {GENERAL, VECTOR, CRYPTO, ...}. A VCPU can simultaneously belong to multiple categories depending on the workload it would run. There are corresponding categories defined for the PCPUs as well, as described in Section 5.3, and they are used to calculate the match value used in Equation (11a). The default category for any VCPU is *GENERAL* and can be updated as desired. The expectations for a VCPU can belong to the set {VARYING_OR_UNKNOWN, MOSTLY_CPU_INTENSIVE,

MOSTLY_CACHE_INTENSIVE, MOSTLY_IO_INTENSIVE, ...} and can be ORed together. The default value, of course, is VARYING_OR_UNKNOWN.

We perform offline profiling of the benchmarks to determine the hints that can be provided to Montage. For example, running spin with vastly different cpu speeds can result in close to 100% slowdown in performance, while running dedup similarly does not affect the runtime more than 10%. Hence, spin is an example of MOSTLY_CPU_INTENSIVE, while dedup is MOSTLY_IO_INTENSIVE.

### 5.4.3 Dynamic Monitoring

There are two parts to the monitoring implemented in Montage. First part observes the behavior of VCPUs by counting performance events like OFFCORE_REQUESTS, LLC_MISSES and RESOURCE_STALLS that indicate a VCPUs bias for the faster or more complex core, as discussed in [50]. We use similar metrics, however, with an implementation in the hypervisor for VCPUs and these counters are measured at each VCPU context switch to attribute values to the right VCPUs. These counters are used to update the $I_r$ values for use by the kinship equations. The second part is to maintain the fault count as seen by a VCPU due to functional asymmetry. We have modified the Xen fault handler to count the faults and migrate the VCPU to a different CPU in the system. A fault lowers the kinship value between a VCPU-PCPU pair as seen from equations in Section 5.3. These parameters are maintained over a moving time window with weights that diminish with age of the parametric value.

## *5.5 System Evaluation*

The experiments shown in this section evaluate the workings of the kinship model by creating various asymmetry scenarios. Since multicore-asymmetric hardware is at a nascent research stage, standard workloads are not available. Therefore, we combine existing application benchmarks to create workloads akin to those seen in clouds and other data-center systems.

### 5.5.1 Testbed

We evaluate the performance of kinship scheduling on multiple machines that exhibit varying degrees of asymmetry.

*SimulatedWM* - is a 12-core Intel Xeon 5645 processor-based machine with 12GB RAM and support for AESNI. It is the platform employed for Usecase2 in Section 5.2. Performance asymmetries can be created by using one or a combination of (i) duty cycle changes – using a hypercall that modifies relevant model-specific registers, (ii) changing the speedstep using Xen's power management interface, and (iii) modifying the LLC size for a a set of cores (using Intel-proprietary technologies). We evaluate the functional portion of the kinship model by turning off the AESNI bit in CPUID, which is how contemporary software determines the presence of AESNI.

For simplicity, we create speed asymmetries with three CPU speeds, either using changes in processor duty cyle (100%, 75% and 50%) or speed steppings (2.4GHz, 2.0GHz and 1.6GHz); we refer to these as fast, medium, and slow cores, respectively. For last-level cache size, proprietary options exist to change it to different values lower than the maximum of 12MB. Experiments described herein use 3MB in order to explore a strongly asymmetric scenario. We refer to the different LLCs as "big" (12MB) and "small" (3MB) cache. The kinship model is not limited to these configurations, however, since the kinship equations use numerical values for processor speed, cache size, and other hardware attributes.

*E5440-5160* - is a prototype dual-socket platform with socket0 consisting of four Intel Xeon E5440 cores @ 2.83GHz, 6MB LLC and socket1 with two Intel Xeon 5160 cores @ 3GHz, 4MB LLC. All cores share the 2GB RAM and remaining resources on the platform. This platform provides a more subtle form of asymmetry due to above properties.

*Xeon-Atom* - is an experimental 2-socket prototype system containing a Xeon X5450 with 6MB LLC and an Atom 330 with 512KB cache. This combination platform, shown in Figure 21.a, has both sockets sharing a 14GB main memory and other resources. The

platform provides a very drastic set of asymmetry combinations.

*Software* - All machines run the latest Xen-4.0 testing source with Linux 2.6.32.41-pvops kernel as Dom0. We used HVM guests running Fedora 12 on SimulatedWM and Fedora 12 paravirtualized guests on the remaining machines due to reasons discussed in Section 5.5.5. The hypervisor code and BIOS are modified to enable booting with hardware asymmetries, as required.

### 5.5.2 Benchmarks

A wide range of workloads offers different resource affinities (e.g., compute intensive, cache intensive, IO intensive). A combination of benchmarks from the SPEC CPU2006 [36] suite, PARSEC suite, hadoop-sort, hadoop-wordcount, and IOZone, are used to create VMs that run representative application scenarios. We use the Intel AESNI sample library implementation [39], which compares Dr. Brian Gladman's AES performance with that of the AESNI optimized library to evaluate AESNI asymmetry (AESbench). In addition, we use the following microbenchmarks: (i) ExtremeLoop (or Spin) - this is a simple and extremely predictable loop benchmark, and (ii) CacheBuster - it reads or writes elements of a byte array in a random order for a given number of iterations, using algorithms that stress the cache in various configurable ways.

### 5.5.3 Experimental Methodology

Microbenchmarks are used to establish correct and predictable scheduling behavior for our kinship equations. Additional application workloads evaluate the performance of kinship scheduling in more complex scenarios. These experiments use scenarios that represent the workload mixes witnessed in conventional cloud and data-center systems.

Evaluations launch a given number of VMs $m$ times and average the time spent, with workloads running within VMs for $n$ times to obtain an average of the metric measured. For the experiments performed, we also present a comparison of the average VM running time for each launch, which is the time spent in running the $n$ instances of workloads for one

launch and the deviation seen. We measure floor and ceiling numbers, wherever possible. These give the worst and best results in an experiment scenario as predicted by say, an oracle that is experimentally (but not provably) optimal. In order to gather these floor and ceiling numbers, we use prior knowledge about the execution pattern of a benchmark on asymmetric components and then pin the VCPUs executing those benchmarks to achieve the best mapping as understood from the static profiling. However, since this mapping is static throughout the course of the experiment, it is likely that the ceiling, for certain cases, might not really show the best performance because it could not migrate some VCPU to the fastest idling PCPU. In contrast, Montage and Xen schedulers would actually perform such migrations in an effort to conserve work (e.g. a fast PCPU could be idling between two consecutive runs of a CPU intensive benchmark). Also, these mappings are across the entire scenario and not at an individual benchmark level. So for example, what is good for an IO intensive benchmark might not be what floor and ceiling achieve. Additionally, since the focus of this chapter is on hypervisor-level scheduling, we pin the benchmarks to guest VCPUs. This helps factor out perturbation due to varying scheduling decisions made by the OS that could disturb the evaluation of the VMM scheduler in the absence of an asymmetry-aware guest OS.

### 5.5.4 Evaluation

We evaluate the Montage kinship scheduler along different dimensions using the benchmarks and testbed described above, combining them in manners interesting to highlight the various aspects of kinship scheduling.

#### *5.5.4.1 Performance asymmetric scenarios*

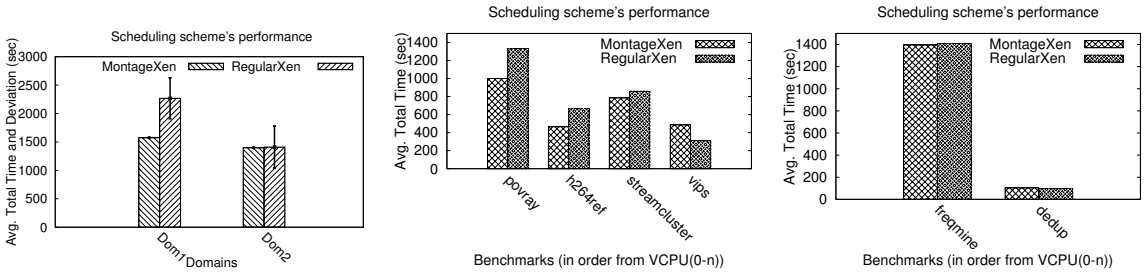
*PCPU speed asymmetry:* Table 3 shows the different configurations used to evaluate CPU speed asymmetry. Figures 25 and 26 show the performance comparison between Montage and Regular Xen on SimulatedWM platform.

*Speed*1 creates a scenario with a media-processing type of VM and a data-mining type

**Table 3:** PCPU speed and cache asymmetry configurations for SimulatedWM (12core, 12MBLLC, 12GB memory

| Name | Benchmarks (VCPU-wise) | PCPU Configuration $\langle \#fast, \#medium, \#slow \rangle$ | Dom0 cores |
|---|---|---|---|
| Speed1 | VM1-Media $\langle povray, h264ref, streamcluster, vips \rangle$ VM2-DataMining $\langle freqmine, dedup \rangle$ | $\langle 2,2,2 \rangle$ | 6 |
| SpeedShare1 | VM1-Media $\langle povray, freqmine, h264ref, vips, streamcluster, dedup \rangle$ VM2-Media $\langle x264, bodytrack, streamcluster, vips \rangle$ | $\langle 2,2,2 \rangle$ | 6 |
| Speedcache1 | VM1 - 1 spin, 1 cachebuster(small) VM2 - 1 spin, 1 cachebuster(large) | $\langle 2,0,2 \rangle$ | 8 |

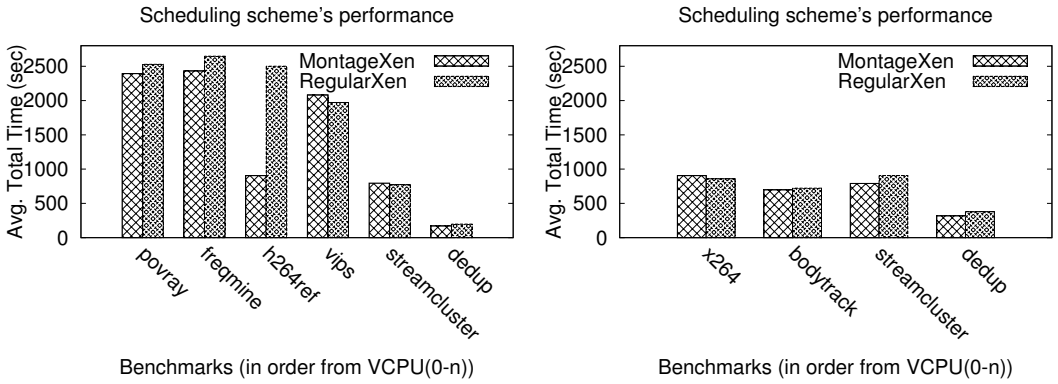
of VM, using corresponding benchmarks from SPEC and PARSEC. These benchmarks vary in their degree of CPU sensitivity and hence, show less vs. more performance variation when run on asymmetric cores. As seen from Figure 25.a), Montage takes about 30% less time for VM1 compared to Regular Xen to complete the entire experiment run, without hurting VM2's performance. It shows highly stable behavior compared to Xen because of its recognition of asymmetry and the predictability of the model that governs scheduling decisions. Further, as seen from Figures 25.b) and c), Montage does well for most benchmarks. Vips and dedup see some performance degradation because they are always executed on the two small CPUs, unless big and medium are free from running the more intensive workloads. Regular Xen on the other hand, schedules workloads without any regard to asymmetry and shows high degradation for benchmarks like povray and h264ref.




a) Total time comparison b) VM1 benchmarks c) VM2 benchmarks

**Figure 25:** Total time and per-benchmark results for Speed1

Figure 26 shows scheduling performance in the presence of cpu speed asymmetry when the system is oversubscribed, using standard benchmarks. We run two media VMs with benchmarks as shown in Table 3. As seen from the graphs, Montage on average performs better than Xen for most benchmarks and significantly so for some of them, like h264ref and streamcluster.

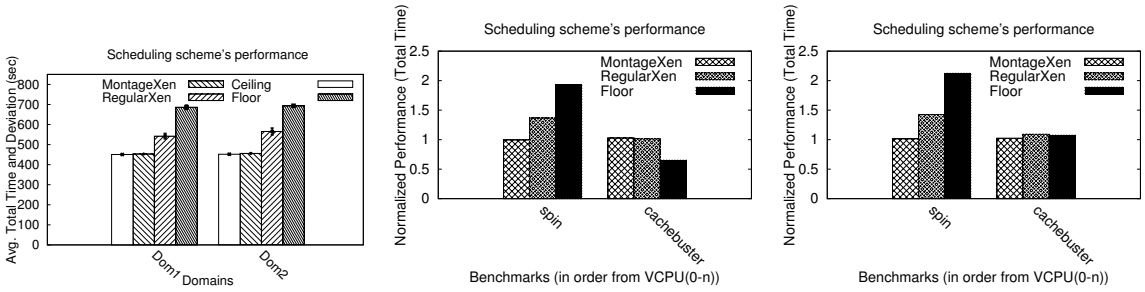



a) VM1 benchmarks b) VM2 benchmarks

**Figure 26:** Per-benchmark results for SpeedShare1

*PCPU speed + cache size asymmetry:* Next, we add the last level cache size asymmetry dimension to our evaluation. Table 3 shows the configuration Speedcache1 used to evaluate CPU speed and socket cache size asymmetry. Figures 27 shows the performance comparison between Montage and Regular Xen. A more generic example with real benchmarks is also evaluated next.

The small cachebuster instance fits just within the 3MB (small) LLC, while the large instance can just fit within the 12MB cache. Figures 27.a)-c) indicate extremely good performance and very low standard deviation for Montage, whereas the absence of asymmetry awareness again hurts Regular Xen. Regular Xen, however, does not show as bad a performance as indicated by the floor numbers because of its work conserving nature. The deviation can be attributed to the fact that every possible schedule for the given VCPUs on the available set of PCPUs is equally likely, and that implies Xen can witness variable performance depending on the mappings created.



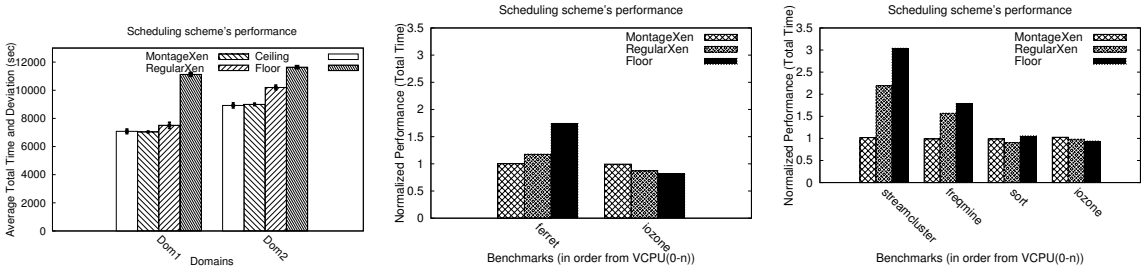

a) Total time comparison b) VM1 benchmarks c) VM2 benchmarks

**Figure 27:** Total time and per-benchmark results for Speedcache1

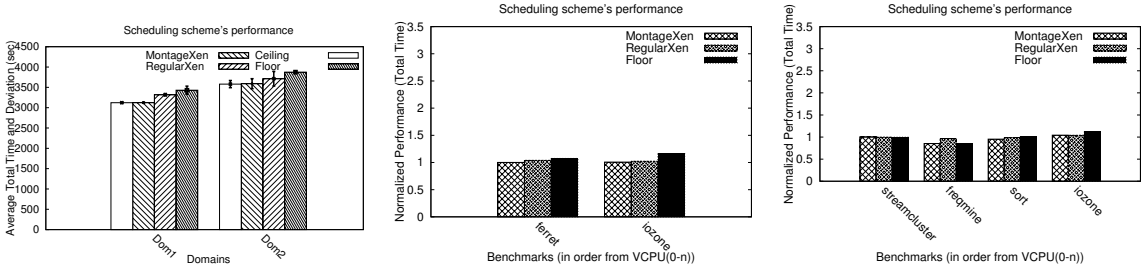
#### *5.5.4.2 Performance with portability*

*Usecase1 performance on all three platforms*: Earlier, in Section 5.2, we used two use-cases to illustrate the intuition behind and definition of kinship. It involved two VMs, VM1 running *ferret* and *IOzone-512M* on 2VCPUs and VM2 running *streamcluster*, *freqmine*, *sort-1G* and *IOzone-512M* on 4VCPUs respectively. Of these, ferret and streamcluster are compute intensive, freqmine and sort-1G are memory intensive while IOzone is disk intensive. We now evaluate this scenario on all of our platform to demonstrate (1) performance improvement achieved on all platforms compared to asymmetry-unaware Xen hypervisor, (2) general applicability of kinship-based scheduling across a wide range of asymmetries and their combinations, and (3) portability of kinship based implementation on shared-ISA machines.

Figure 28 shows the results of performance comparison between Montage scheduler and Xen scheduler. As we can see, appropriate scheduling of workloads on all machines leads to a definite performance improvement for MontageXen compared to RegularXen on all platforms. On Xeon-Atom, where the VMs have 2 Xeon and 2 Atoms available to share, MontageXen shows low variability and an overall VM performance improvement of 6% and 12% respectively compared to RegularXen. MontageXen is comparable to the ceiling numbers shown in the figures. Compute intensive benchmarks like ferret and streamcluster see a stark performance difference from better mappings performed with kinship metrics. Since E5440-5160 shows less drastic asymmetries, the performance difference is




a) Total time comparison | b) VM1 benchmarks On Xeon-Atom platform | c) VM2 benchmarks (drastic asymmetry)




a) Total time comparison | b) VM1 benchmarks On E5440-5160 platform | c) VM2 benchmarks (subtler asymmetry)

**Figure 28:** Total time and per-benchmark results for Usecase1 from Section 5.2

less striking. However, the results show low variation and better overall performance. The Dom0 in VCPUs in this case share all 6 PCPUs with the two VMs. Montage sees similar performance benefits even on SimulatedWM platform too.

#### *5.5.4.3 Efficiently addressing functional asymmetry*

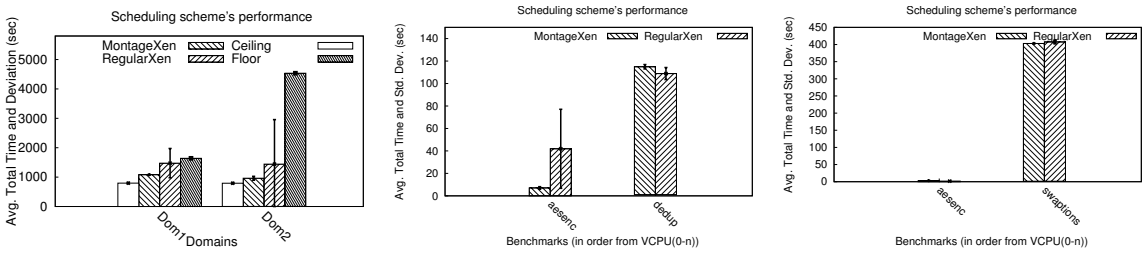
*PCPU speed + AESNI functional asymmetry:* we evaluate Usecase2 from Section 5.2 for the functioning of the kinship equations in the presence of functional asymmetry. As explained there, we choose the AESNI instructions on Xeon processors for this purpose. The configuration created is shown in Table 4. The AESNI benchmark allows us to choose the number of blocks and loops. We keep the number of blocks constant, but vary the loops for big vs. small AES instances. The benchmark checks if a CPU supports AESNI using CPUID. If it does, then the hardware version of AES is executed, else the software version is chosen. The software version is comparatively much slower than the hardware counterpart, as expected. We use the AES-CTR encryption with a 128-bit key in this example.

Figures 29.a)-c) compare Montage and Regular Xen and demonstrate the performance

**Table 4:** PCPU Speed + AESNI Asymmetry Configuration

| Name | Benchmarks (VCPU-wise) | PCPU Configuration $\langle \#fast, \#medium, \#slow \rangle$ | Dom0 cores |
|---|---|---|---|
| SpeedAes | VM2 - 1 AES (large), 1 dedup<br>VM1 - 1 AES (small), 1 swaptions | $\langle 2,0,2 \rangle$ | 8 |

advantages derived from the awareness of functional asymmetry in Montage. While the work-conserving nature of Xen is quite useful in improving performance, there is a really high deviation witnessed by Xen due to occasional execution of the aesenc-large instance on a non-AES supporting CPU leading to a 2X performance improvement seen by Montage for VM1. The predictability of scheduling in Montage in the absence of over-subscription is a very important property that establishes its prominence compared to the current Xen scheduler.




a) Total time comparison b) VM1 benchmarks c) VM2 benchmarks

**Figure 29:** Total time and per-benchmark results for SpeedAES

#### *5.5.4.4 Stability analysis*

All the previous results show vary low standard deviation for MontageXen. Now in order to evaluate how kinship scheduling reacts when a stable state is perturbed, we perform the following experiment. We run VM1 with six workloads (*ferret, streamcluster, freqmine, sort-500M, iozone-64M, iozone-64M*) on the Xeon-Atom platform for about 2600sec. The PARSEC workloads here use their simlarge dataset. These benchmarks are launched repeatedly in a loop until the VM is alive. Its important to note that, of these benchmarks,

the first two are compute intensive, next two are memory intensive and last two are IO intensive. After 400sec of VM1 execution, we introduce VM2 with two compute intensive (*blackscholes, swaptions*) with native dataset, a cache intensive *hadoop wordcount-150M* and another *sort-1G*. This VM is run for about 800secs, then just VM1 for 600sec, again followed by another VM running (*canneal, dedup, freqmine* with native dataset and *iozone-64M*). The VMs introduced to perturb affect different benchmarks in VM1 due to the nature of their benchmarks. Figure 30 shows the time chart as seen from benchmarks in VM1 during this experiment. The points represent midpoint of the start and end times of one run of any benchmark. The numerical values to the right indicate the number of times each benchmark managed to complete a run during VM1 lifetime. MX is MontageXen and RX is regular Xen.

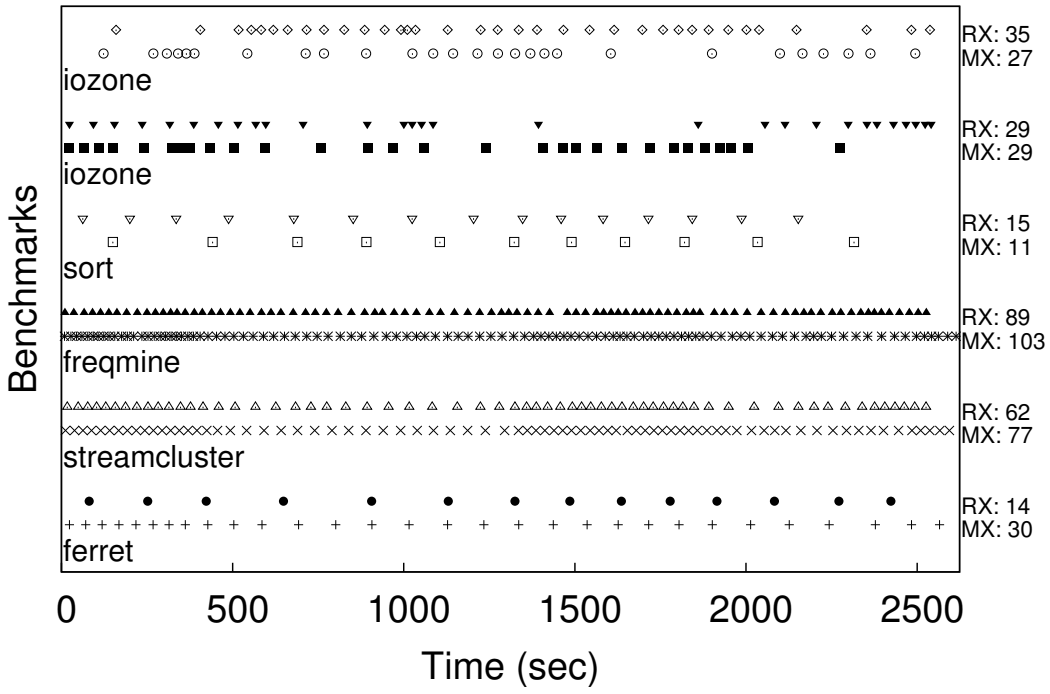



**Figure 30:** Kinship scheduling (MX) results in more predictable and regular domain runs compared to RegularXen (RX) even when perturbed occasionally

The figure clearly indicates how the smoothness of execution with Montage (due to its appropriate mapping of workloads to suitable PCPU type) and stability of kinship equations results in greater overall work done by VM1 by launching the benchmarks more number

of times than RegularXen. The pattern also indicates an even spacing out of workloads in the presence and absence of perturbance. The behavior of IO workloads is slightly less predictable due to other factors like disk and Dom0 IO handling limitations coming into play. But the compute and memory intensive benchmarks execute with a much better performance in Montage.

#### *5.5.4.5 Scalability analysis*

Next, we evaluate the overhead introduced by the additional computation for kinship, in different portions of the hypervisor code on the SimulatedWM platform. The overhead of VM creation and destruction remains within 14-24$\mu secs$ irrespective of the existing number of VMs in the system. VCPU creation and destruction costs 3-8$\mu secs$. The cost of assigning a VCPU to a PCPU varies between 1.5-8$\mu secs$ for PCPUs between 4 to 12. Thus, most functions show very little overhead. However, the most important component of kinship scheduling is the rematching algorithm, Algorithm 5 that runs periodically to revises VCPU-PCPU mappings based on change in number of VCPUs or updates to the kinship parameters occurring due to observations or change in platform characteristics. Figure 31 shows the behavior with increasing numbers of VCPUs at different number of PCPUs in the SimulatedWM system.

As seen from Figure 31, this function increasingly adds to the overhead as the number of VCPUs or PCPUs increase. However, reassignment is carried out at most once every four Xen accounting cycles (so 120msecs) and hence, even with 180$\mu secs$ overhead at 48 VCPUs and 12 PCPUs, it is less than 1% of the accounting cycle time. The graph also indicates that VCPU count becomes the dominating factor at higher values VCPUs.

Now consider the fault-and-migrate case where a sVCPU from some VM runs on a core that does not support the full x86 ISA. This results in a fault when an unsupported instruction is executed, and the hypervisor then schedules the sVCPU onto a different core (one that supports the instruction) so that it can continue its execution. Such micro-scheduling

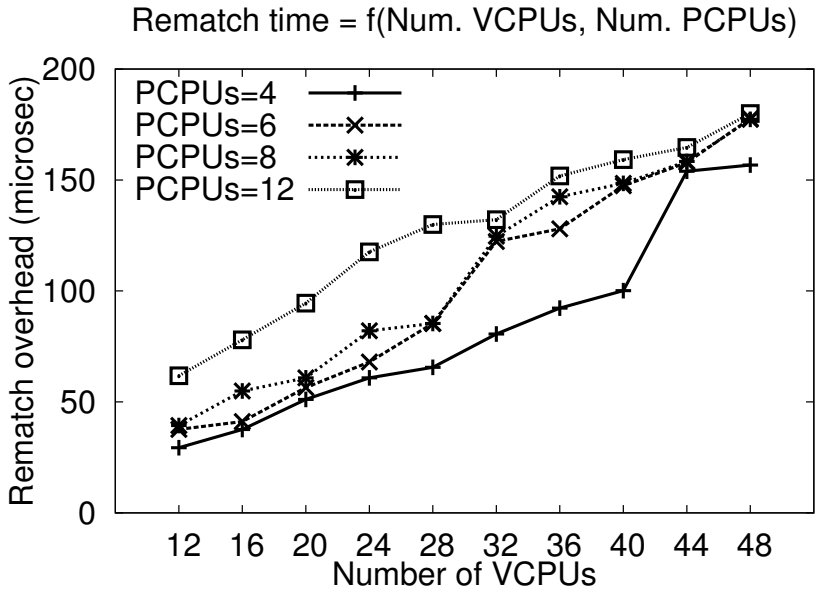



**Figure 31:** VCPU-PCPU rematching overhead

actions occur outside the regular scheduling interval used by the hypervisor or OS and thus, must be highly efficient in order to cope with VMs that frequently use diverse sets of unsupported instructions. We have evaluated the overhead for such faulting on unsupported instructions to be $20\mu sec$ as measured from user level, and average VCPU migration cost in Xen is measured to be $495nsec$. The fault handler simply updates the fault count for the VCPU-PCPU pair and does not cause any kinship modification within the handler. Those are handled at the next rematch.

#### *5.5.4.6 Parametric knobs for kinship and learning from observations*

Equation (3) shows the weights that can be used to modify the effect that each kinship component can have on the overall kinship value. Section 5.4 talks about how we can observe faults and migrate a VCPU to handle functional asymmetry. As seen from Equation (11), the functional component of kinship between some VCPU and PCPU pair becomes increasingly negative as the fault count increases. At some point, the functional component becomes negative enough to offset any positive value from the performance kinship component due to other resource matches. That's when the VCPU is scheduled on other PCPUs. Since it's a moving time window, the VCPU may eventually get migrated back to

the same PCPU if other factors permit.

We execute two VMs with two VCPUs each on the E5440-5160 platform to evaluate the effect of weight and functional kinship. VM1 runs *ferret* and *sse_mat* on its VCPUs while VM2 runs *streamcluster* and *sse_mat*. *sse_mat* is a microbenchmark that repeatedly runs SSE4.1 based matrix multiplication on really small matrices. Both *ferret* and *streamcluster* are more CPU and cache intensive than the *sse_mat* instances. We allow the four VCPUs to run on any of the two 5160 cores that do not support SSE4.1 as well as two out of the four E5440 cores that do support SSE4.1. We do not indicate any category expectations for the VCPUs. Hence Montage initially picks the two benchmarks from PARSEC, *ferret* and *streamcluster* to run on the E5440 cores due to their higher speed factor and larger LLCs. This leaves the 5160 cores, without the SSE4.1 support, for the two instances of *sse_mat* microbenchmark. This will obviously lead to faults since the benchmark expects SSE4.1. Montage maintains this fault count before migrating the VCPUs. However, the VCPU can be migrated back until its kinship value precludes it from doing so, as discussed above.

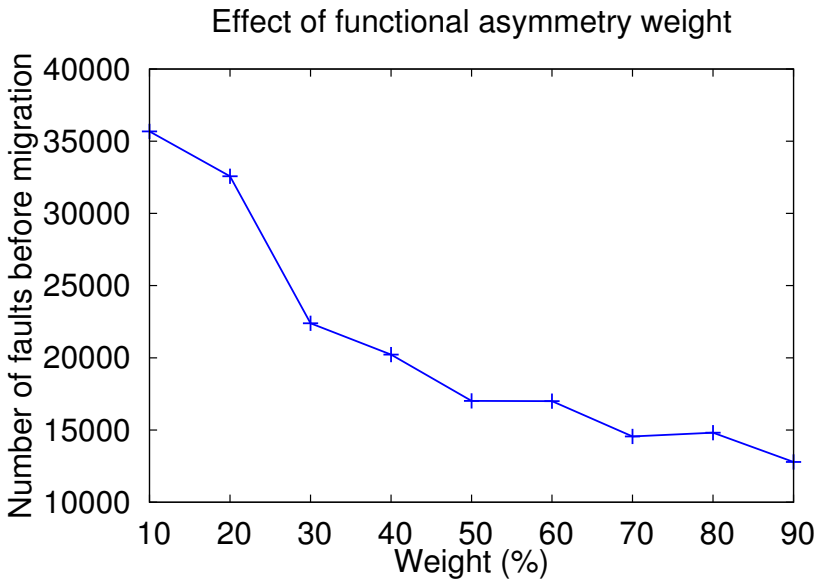



**Figure 32:** MontageXen migrates a faulting VCPU from the corresponding PCPU and its reactiveness can be modified through the weight assigned to functional portion of kinship

Figure 32 shows the effect of modifying the kinship weights on the maximum fault

count seen by VCPU running our SSE microbenchmark. The fault count in this example reaches a high value before the VCPU is permanently migrated due to the performance kinship match which requires a larger negative value to offset the effect. This behavior can be changed using a fault threshold that leads to the VCPU classification as a VECTOR VCPU which will affect the *match factor* defined in Section 5.3 in Equation (11). The migration is also affected because the other VCPUs, with higher kinship values, get to choose their PCPUs before the VCPUs running the SSE microbenchmarks.

### 5.5.5 Discussion

**Performance:** as seen from the evaluation above, kinship-based scheduling performs efficiently on asymmetric platforms, with low standard deviation for application runtimes. The stable nature of scheduling can be of extreme importance in HPC environments, despite a current lack of support for sophisticated runtime observation. The kinship equations enable a dynamic scheduling environment for current and future asymmetric platforms due to their explicit incorporation of various asymmetry parameters. The flexibility of being able to incorporate different parameters is important due to the evolving nature of asymmetric platforms. As monitoring becomes finer grained, this model will also adapt well to applications running in a cloud environment, even without hints.

**Implementation challenges:** are presented by the different generation cores possibly supporting different revisions of instructions, for example, we had to modify Xen to boot on the E5440-5160 platform since the two sockets supported different revisions of the VM control structure (VMCS) introduced by VT-x support on the Intel cores. In order to use HVM guests on this platform, we could have had to read and write VMCS each time a VCPU was migrated from one generation core to the other, in software. This would have led to additional overhead. We did not evaluate this overhead or implement this migration, because it can be safely assumed that Intel would provide hardware or micro-architecture

conversion to support asymmetric migration if such platforms were manufactured as mainstream products.

**Guest scheduling:** the consideration of guest scheduling is out of scope for this research, and we avoid the issue because in experiments, application threads are pinned to VCPUs. However, it is important to consider the ramifications of guest scheduling on the hypervisor scheduler. For instance, a smart guest using the collaboration/hint interface could provide hints to the hypervisor and help it switch VCPU-PCPU mappings based on the kinds of workloads about which it is aware. In fact, such a guest itself could understand asymmetries and try to schedule threads on the right kind of cores. In the absence of such guests, we have to rely on observation to detect phase changes introduced by thread scheduling. With better observation capabilities, kinship scheduling will more easily adapt to new characteristics of VCPUs.

**Fairness:** the kinship equations have a *fairness token* that can be circulated between VMs to ensure higher kinship values for fast CPUs for the VM holding the token. But, we disable this token for our evaluation since we want to highlight the matching of VCPUs to PCPUs based on workload characteristics.

**Hints vs. observation:** what if the hint given for a VCPU is wrong, especially when you can observe? Such cases could be handled by setting up a monitoring and feedback loop to correct observed behavior of workloads, which is also beyond the scope of this dissertation. Kinship equations assume the correctness of hints, and discrepancies observed can be reported to a module outside the performance critical scheduling code.

## *5.6 Related Work*

**Complementary OS research.** Several research efforts have made operating systems asymmetry aware. Asymmetry aware OS scheduling is complementary to kinship-based hypervisor scheduling: it will likely further enhance performance by scheduling 'smartly' at the guest level. Efforts like [53] have focused on processor speed asymmetry, arguing

for fewer big cores and more small cores in the same die area to support high throughput in the system and for accelerating serial phases of parallel applications. In a similar vein, Saez et. al evaluate a comprehensive scheduler [91] that employs efficiency specialization by mapping CPU intensive tasks to complex cores by techniques that involve measuring LLC misses and thread level parallelism (parallelism-aware). The idea is to map highly parallel phases to a larger number of small cores but move bottleneck serial phases to the fast core. Previous work [92] also evaluates the use of application signatures, with information like memory intensity, to assist in matching threads to fast vs. slow cores. [63] explores different policies like fastest-core first, as well as NUMA-aware load balancing for better utilization of fast cores in a fair manner. [58] proposes longest job and critical job to a fast processor first policies. Finally, in [65], the authors offer another fair-share policy, called DWRR, for transparent execution and fair sharing of overlapping or shared-ISA asymmetric systems. However, rather than considering threads to be uniform, kinship directly addresses workload diversity while performing the mapping to underlying platform resources. Finally, while all of these efforts explore pertinent issues with respect to asymmetric systems, they consider only weak forms of performance asymmetry, typically focused on parallel applications or other application components that expect their fair shares of fast cores.

'Bias' [50] is a new metric that enables measurement of application affinity towards fast cores even in the presence of micro-architectural diversity. That work evaluates a stronger form of performance asymmetry, and we use their ideas to add observation capabilities to the Montage hypervisor. However, the kinship metric, coupled with the fault-and-migrate model used in this chapter, supports a larger range of asymmetries and can be easily adapted to an evolving spectrum of asymmetric systems.

**Hypervisor asymmetry research.** The asymmetry-aware scheduler in [44] for hypervisors addresses CPU speed asymmetry and implements fair sharing of fast cores among

VMs, occupying fast cores first when performance is the top concern and prioritizing access to rare fast cores based on VM priorities. While this is the a step towards making hypervisors asymmetry-aware, that work does not consider asymmetry in workloads – a prime motivation behind such platforms. It also does not consider other forms of platform asymmetries like asymmetry in cache sizes or functional units. At the same time, with kinship scheduling, fast cores will also be occupied before slow ones, depending on workload demand, and when all workloads are alike, kinship scheduling behaves like AASH. Kwon et.al. proceed a step further by adding an asymmetry-aware 'active' Xen scheduler [56] that can monitor VCPU behavior to correctly match VCPUs to the underlying speed-asymmetric platform. Compared to these efforts, Montage kinship-based scheduling provides a general framework easily adapted to different platform asymmetries and configurations.

**Helpful techniques.** Differences in contentiousness vs. sensitivity of applications are expressed by a set of equations presented in [102] and validated with experimentation that provides methods to measure indicators to classify applications in order to mitigate memory resource contention. Similarly useful for monitoring support to be added to Montage are the methods in [48]. The symbiotic execution of multiple layers in the software stack in [31, 66] would of course, further enhance convergence to some preferred state, given that different layers will be able to work with more information than otherwise.

## *5.7 Conclusions and Future Work*

Montage is a framework for managing the resources of asymmetric multicore platforms. Its *kinship model* extends the hypervisor with the first complete representation to address both performance and functional asymmetry, in a manner extensible to a wide variety of asymmetric platforms. The model is evaluated with a wide spectrum of asymmetries and workloads able to exploit such asymmetries. The version of Montage realized in the Xen hypervisor enhances Xen's credit based scheduling with the kinship equations described in

the chapter. As shown in the evaluation, the kinship scheduler is superior to the existing Xen scheduler, for a substantial range of platform asymmetries and usage scenarios. The kinship model lays the foundation for more sophisticated research in dynamic asymmetric scheduling.

**Thesis discussion:** Through Montage, we have shown how it helps to clearly identify the personality change in a virtual machine and schedule it to match the type of core that it can run well on. As we have observed in case of the Pegasus model, personalities are fixed and do not change unless we use binary code rewriting or recompilation—currently not under hypervisor control. However, personalities in Montage are more fluid and techniques like fault-migrate or fault-emulate, also discussed in Chapter 3, can enable a runtime personality change. As seen from our experimental evaluation, kinship equations smoothly handle such personality differences and the scheduling logic always tries to match the best type of core for all VCPUs or sVCPUs in the system. We reiterate the importance of recognizing the ISA differences and considering workload characteristics while making system-wide scheduling decisions in order to achieve both, platform utilization and workload performance goals.

**Future work:** There are several research challenges in tuning the kinship computation itself. It was beyond the scope of this work to evaluate the effect of the various weight vectors used in the kinship model. However, they can offer very powerful tunables for adapting the behavior of kinship scheduling to meet diverse or even dynamically varying system goals. For example, increasing the weight of the functional component of kinship could make it more sensitive to instruction fault and it might even result in the replacement of a compute bound VCPU from a compute intensive PCPU if that was the only suitable option for the faulting VCPU. Another very intriguing dimension is that of *self learning kinship models*. While we have focused on individual kinship parameters that could change with changing workload phases or platform asymmetries, it is conceivable that the model (basically the set of equations) could themselves be adaptable. A simple

way to do this is to implement kinship equations aimed at achieving different system goals in separate resource scheduling domains and switching VCPU from one domain to another based on its changing expectations. Another way for these equations to adapt is by way of dynamic weight-vector changes that could be triggered by different events in the system like performance degradation due to increased number of instruction faults could lead to increase in the weight of the functional kinship component.

Our ongoing research is implementing methods to enhance the dynamic characterization of workload behavior, in order to provide sophisticated, runtime input for certain parameters included in the kinship equations. The goal is to accurately detect phase changes in workloads and hence, enable consequent re-mappings of VCPUs to PCPUs. This could in fact benefit from history-based predictions and machine learning algorithms that could use input from performance counters and past values to predict the phase changes. Future research should evaluate the approach with asymmetry-aware guest operating systems and with more sophisticated hint-channels, perhaps even allowing guest VMs to explicitly state their current requirements. It should also consider extensions to the kinship model that deal with entirely non-overlapping ISAs, as with x86 vs. graphics processors, by expanding on the functional component of kinship. In the Montage evaluation, we have focused on achieving high performance for the workloads. However, this goal could be changed to other system goals like achieving low power. The equations would then have to be adapted as such. Also of interest are scalable algorithms for mapping VCPUs to PCPUs, for future multicores in which the number of PCPUs could be in the 100s and the number of VCPUs could be in the 1000s. The complexity of the current rematching algorithm is proportional to the number of VCPUs and PCPUs in the system and all kinship values are recomputed each time this algorithm is executed. However, it might be possible to calculate and accommodate only those kinship values that have changed parameter values. Scalability could also be achieved by introducing hierarchical solutions in which VCPUs are passed through multiple kinship-based schedulers. The cpupool component of kinship

can be instrumental in defining allowable PCPU zones for the VCPUs based on say different coherence domains. This could restrict the complexity along the PCPU dimension and make the computations faster.

# CHAPTER VI

# ATTAINING SYSTEM PERFORMANCE POINTS: REVISITING THE END-TO-END ARGUMENT IN SYSTEM DESIGN FOR HETEROGENEOUS MANY-CORE SYSTEMS

While existing and upcoming management methods routinely leverage system-level information available to the hypervisor about current and global platform state, we argue that, for future systems, there will be an increased necessity for additional information about applications and their needs. The Montage architecture specifically incorporated user-provided information about workloads in the scheduling formulation. Its evaluation highlighted the benefits that could be achieved from the simplest form of hints like VCPU execution category or CPU expectations. This chapter considers 'performance points' as a general interface between the virtualization system and higher layers like the guest operating systems that run application workloads. Building on concrete examples from past work on APIs with which applications can inform systems of phase or workload changes and conversely, with which systems can indicate to applications desired changes in power consumption, we first define and then show how performance points are an effective way to better exploit asymmetries and gain the power/performance improvements promised by heterogeneous multicore systems.

Large-scale applications driving heterogeneous hardware developments go beyond single parallel programs to include complex codes that require multiple forms of processing, ranging from network-centric servers running 'front matter' code as in web servers, to application servers carrying out computationally expensive tasks that prepare and process content, to data-intensive backend services interacting with storage. An interesting example is a web application with strict latency requirements, like the financial code we will

describe in Section 6.1 of this chapter. This code has transactional components that interact with clients and process input data, computationally intensive components that determine new option prices and in addition, encrypt data before returning it, and logging components that interact with storage. Its processing requirements include rapid data aggregation and transmission, intensive scalar and vector computation, and specialized processing like that needed for encryption. Its degree of parallelism depends on the number of clients and options being processed, and its workload is dynamic due to client load variations. Such varied requirements make codes like these excellent candidates for heterogeneous platforms that promise improvements in the power-to-performance ratios seen by service providers and end users, and this is even more so the case under conditions of workload and server consolidation in data center systems.

Extensive ongoing research and development are aimed at realizing the potentially superior power/performance properties offered by upcoming heterogeneous platforms. Source codes and libraries are being rewritten to take advantage of accelerator chips, as evidenced by coding efforts involving the CUDA and OpenCL runtimes and APIs exported by GPUs [104, 23]. Tool chain developers are creating methods by which to identify application components suitable for acceleration, with early work focused on graphics codes and more recent work addressing vectorizable and high performance codes [100, 88]. Such efforts are supported by runtimes that offer essential functionality for creating efficient application codes able to leverage diverse underlying accelerator architectures: (1) they provide portability through intermediate program representations that can be translated to efficient codes for specific target accelerators [17], such as the PTX assembly used by the CUDA runtime [77], and (2) they can act as runtime agents able to gather information about application performance on target processors, determined by dynamic factors that include input or data sizes, the frequency of interactions between accelerated vs. scalar computations, latency vs. throughput requirements, and others [19, 18]. Finally, there has been substantial work

on system-level methods that manage the resources presented by asymmetric [64] or heterogeneous [30] platforms. This includes methods focused on certain platform properties, such as their parallel nature coupled with NUMA memory or NUCA cache architectures [26, 91] and methods concerned with efficiently scheduling the different types of processors present on such platforms [68, 64, 91, 50].

The ongoing work mentioned above indicates the benefits of involving the entire software stack—from application codes, to compilers and libraries, to runtimes, to operating systems—when meeting the service demands of applications with heterogeneous underlying platform resources. For example, we note that for CUDA codes, compilers are able to determine the code components suitable for execution on accelerators [100], but they lack information about actual input sizes, execution frequencies, or performance effects due to different ways in which multiple codes are sequenced and/or share accelerator resources. In addition, while there is proven value in using both accelerator and scalar cores when running parallel applications [68], decisions as to where which codes are run will depend on current workloads and resource availability. Further, a runtime decision to run say a PTX code on a scalar core requires toolchain use for just-in-time code generation. Finally, for parallel applications that exhibit phase behavior, there may be may be different degrees of parallelism in different phases. Application runtimes can recognize such behavior and can share information about the needs of future phases with system-level resource management [19, 89].

The important lesson and the key insight derived from past work and driving our research is that *in order to most efficiently utilize heterogeneous platforms, there should be transparency—not opacity—between the different levels of abstraction present in modern systems*, including applications, toolchains, runtimes, systems, and hardware. Specifically, there should be ways for applications and runtimes to inform systems about known or inferred needs and requirements [51, 87, 73, 54], and there should be platform-provided information which systems can use to better manage platform resources. Common methods

concerning the latter are the runtime use of hardware performance counters or system-level monitoring methods [61] and similar arguments can be made about runtime information provided to toolchains in order to fuse or merge fine-grained parallel computations into larger units better suited for execution on platform cores or caches [29].

System virtualization contributes to the need for transparency and information exchange across levels of abstraction. This is because when virtualized, guest operating systems embedded in virtual machines (VMs) cease to have complete control over the actual hardware platforms on which they run, being restricted to work with the virtual platforms exposed by hypervisors or virtual machine monitors (VMMs). Guest VMs, however, have internal methods that actively manage underlying resources, such as CPU scheduling, memory usage (e.g., NUMA memory-aware operating systems), disk scheduling, network usage, and resource management for improved energy efficiency. It is important, therefore, to present to guests and applications the heterogeneous platforms with exactly the characteristics they expect to see, and conversely, to enable the management methods resident at different levels of abstraction to operate 'symbiotically' and jointly to gain desired performance goals [59]. The need for and usefulness of such information sharing and interaction has been demonstrated for broad classes of systems and application domains, including in [86, 73, 59, 51, 46], with one intuition being that resource management methods running at different system levels have differing purposes, such as VMM-level resource management concerned with platform-level properties like achieving high platform utilization vs. VM-level management focused on meeting the demands of applications run in each VM.

The traditional 'end-to-end' argument, classic design principle proposed by Saltzer et. al. [93] for networking and later adopted by many other system designs, suggests that: "The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the end points of the communication system. Therefore, providing that questioned function as a feature of the communication system itself

is not possible. (Sometimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement.)" Or in other words the principle, called the end-to-end argument, suggests that functions placed at low levels of a system may be redundant or of little value when compared with the cost of providing them at that low level. In summary, this chapter states and motivates the following, somewhat contrasting 'end-to-end' argument for heterogeneous system design: *"for future heterogeneous platforms, obtaining high levels of efficiency measured with needs-centric application metrics and system-centric metrics like utilization/performance, requires cross-layer interfaces that support rich, online methods for behavior understanding and management."*

Given this argument, the abstraction we introduce for these purposes is that of ***Performance Points***, which can be thought of as a tuple of $\langle GoalSet(G), DataSet(D), Conduit(C) \rangle$ where

- the Goal Set derives its parallel from 'set points' [22] in control theory defining desired performance metrics,
- the Data Set consists of data or information associated with achieving a particular Goal in G, and
- the Conduit is the passage or channel that implements the interfaces between different levels of abstraction for exchange of information in D.

In our abstraction of performance points, G can range from being a value for some parameter, e.g., expected throughput, to utility functions provided for use by system-level management [16], to a threshold on some metric (e.g., maximum permissible latency), to the empty set where the goal is not definable. D consists of data exchanged between layers in order to reach some goal or to simply provide information, like performance counters, expected CPU category, etc., that can later be utilized by one of the layers. C is the channel through which such data is exchanged, and it can consist of a set of API calls like the ones defined in Section 6.3, a shared memory area polled by some layers,

e.g., Xen store [14], or an event channel [55] providing active notification support. For the heterogeneous platforms that are the focus of our work, we identify multiple important performance points:

***App points*** – are the performance points defined between applications and the OS/runtime layers. They consist of goals set by applications to inform underlying layers about desired service level agreements (SLAs), such as the latency requirements seen by financial applications like the one explained in Section 6.1. Responses seen by applications may be continuously-provided monitoring information about achieved performance, upcalls informing applications about requirement violations [15, 55].

***OS points*** – occur at the boundaries of operating systems and hypervisors. They may be used to meet a platform-level goal, e.g., to execute within a certain power budget, where data D could be the instruction by an operating system's power regulator to the hypervisor to set its processor to a certain power state, which may then cause the hypervisor to take a management action that tries to honor this request [73]. They could also be used to provide information about OS-level resource usage, such as its numbers of threads or its memory usage, which enables the hypervisor to make resource management decisions [112]. Conversely, there is useful information hypervisors can provide to systems about their virtual platforms, such as its memory characteristics [86] or the status of its available processing resources [30].

***Platform points*** – are the abstractions used by the hypervisor to interact with the physical platform under its control. This includes the 'up' information provided by performance counters exported by hardware and exposed to/used by the hypervisor, and 'down' controls like the hardware methods used to place processors into certain power states.

Our contributions to the "end-to-end argument" can be summarized as follows:

- performance points implement the cross-layer interactions needed to fully realize the power/performance potentials of heterogeneous systems and platforms, and
- their use guides the resource management decisions made at different levels of the

systems stack.

- We demonstrate this with experimental evidence collected on asymmetric computing platforms.

The next section describes the end-to-end argument and further defines performance points. We discuss a plausible future software architecture in Section 6.2 followed by the APIs realized by performance points in Section 6.3. Related work appears in Section 6.4, followed by conclusions and future work.

## *6.1 Revisiting the End-to-End Argument*

Re-iterating the end-to-end argument by Saltzer et al. [93]: "The principle, called the end-to-end argument, suggests that functions placed at low levels of a system may be redundant or of little value when compared with the cost of providing them at that low level". Our work draws an alternate interpretation of this statement to mean that it may be inefficient to provide functionality without knowledge about how it will be used, and conversely, that it is hard to implement such functions without sufficient knowledge about underlying system properties. In particular, when dealing with heterogeneous hardware and for the example of the virtualization layer, that layer cannot implement its required property of proportional sharing for the virtual CPUs (VCPUs) used by guest VMs (1) without understanding how guest VMs desire to use VCPUs, and (2) without exporting platform heterogeneity to guests so that they can best exploit their capabilities.

As noted in this chapter's introduction, (1) and (2) above are borne out by previous work, including recent results termed 'symbiotic' virtualization [59] for the high performance domain, which defines interfaces that permit hypervisors to better understand and thus, better support the memory structures assumed by guest VMs. Complementary work by Rao et al. [86] shows the importance of exposing physical NUMA memory structures to guest VMs. Further, Nathuji et al. [73] discuss the implications of sharing information between the VMM and guests for effective system-level power savings and Uhlig et al. [107]

observe performance improvements by providing locking hints. Additional empirical evidence concerning API exposure exists for real-time and adaptive systems, in terms of their use of utility functions and timing specifications, for I/O scheduling [46], and others.

More concretely, consider the end-to-end argument with respect to our representative latency-sensitive ICE application [69] running on a heterogeneous multicore platform. This application receives and then processes client requests, e.g., option prices, then encrypts its responses and returns them to clients. The application is based on a pull-push model, where the client "demands" data by subscribing to certain feeds from the server. As the pricing fluctuates, the server "pushes" the changes out to the client, within strict timing constraints. It is structured in 3 tiers – client-facing web servers, application servers, backend servers – with many-many relationship between tiers, where the processing performed in these tiers may be characterized as follows: (1) send/receive request, price changes – response or network phase, (2) compute option prices for received data – processing/compute intensive phase, and (3) encrypt user responses – security or encryption phase. There is also some intermediate processing to aggregate data from different options/futures prices to relay to the client, constituting a combined computational and data phase.

Most complex enterprise codes have different stages or processing phases [82], and it is precisely the presence of such diversity that drives our study of the utility of asymmetric hardware platforms. With the specific platform used in our work, to meet the needs of the financial application described above, for instance, its fast and slow cores can be used to satisfy the processing vs. network phases of this code, and the cores supporting, e.g., Intel's advanced encryption instructions are useful to the code's encryption phase. This also means, of course, that the potential power/performance advantages gained from such asymmetries will be realized only if the hypervisor maps the right VCPUs onto the right cores—asymmetric scheduling—and to assist such scheduling, interfaces are needed between the application/OS and hypervisor system layers. Such interfaces are provided by the performance points advocated in our work.

### 6.1.1 Defining Performance Points

Figure 33 uses the ICE application outlined above to depict both the performance points described below and the information exchanges they support. In the following description, we give examples of Goal Sets and Data Sets at the performance points listed earlier but defer the discussion of the conduit or channel portion of the performance points tuple to Section 6.3.

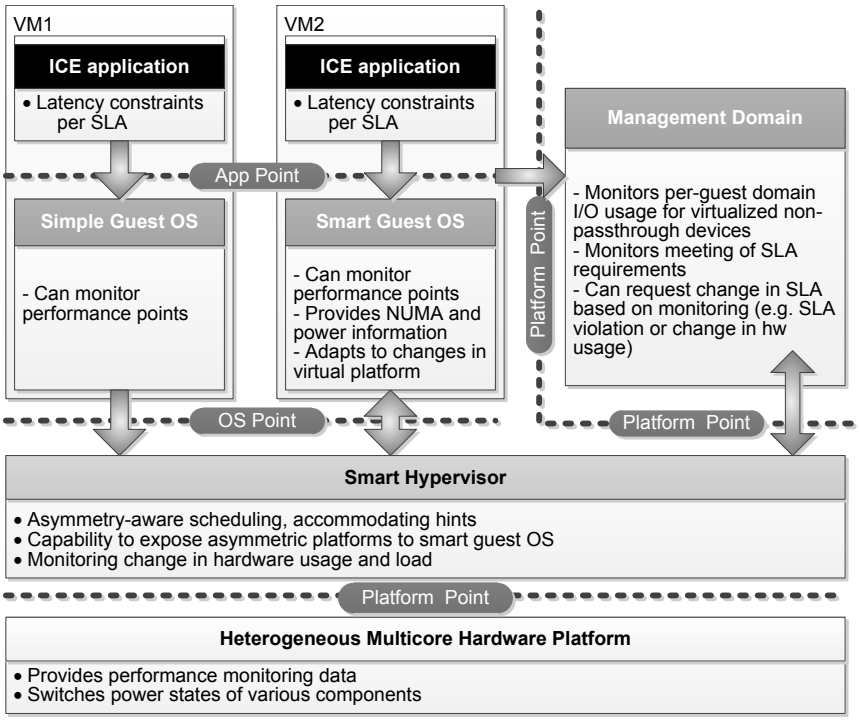



**Figure 33:** Performance Points

**App Points**: most applications are written using programming models suitable for a wide range of potential hardware, and hence, it is unreasonable to expect them to explicitly convert their performance expectations to the corresponding platforms on which they run. They may convey processing or parallelization expectations via annotations that can help with code generation, and they may state runtime requirements in terms like SLAs, but they cannot be expected to articulate all of the potential runtime behaviors, such as their expected processing times dependent on dynamic input data types or sizes or concurrent workload in the system. For instance, an app point suitable for the ICE application described above could include SLAs like response rate or latency per request in the Goal

Set G, which can serve lower system levels as indications about whether the application is performing as expected and the annotations could provide information for the Data Set D.

**OS Points**: the OS manages resources on its virtual platform. In addition to "passing through" to the hypervisor information like number of violations (i.e., SLA violations) observed for specified app points, there are OS-centric metrics like the observed number of page faults or number (and causes) of context switches that can be used by hypervisors to determine suitable resource allocations to guest VMs, all of which constitute useful parameters for the Goal Set of OS Points.

The Data Set, however, remains a subject of ongoing research. Previous work on asymmetry-aware guest OS schedulers [48, 50, 91, 53], for instance, identifies the OS extensions needed to make an operating system aware of asymmetries, to gain performance and/or power improvements. Also relevant is prior work on scheduling methods based on cache affinity [105]. In all such cases, since the OS learns about and/or acts in accordance with certain aspects of the application, it is easily conceivable that it can also convey such information to the hypervisor, perhaps when there are changes in application phases from I/O to compute or from cache-agnostic to cache-intensive. The hypervisor can then use such information to better utilize the underlying asymmetric computing platforms it provides to guests. For example, for our financial application's send/receive phase, knowledge about the fact that the application is currently in the network phase could be conveyed to the hypervisor, which in turn could use it to move the corresponding VCPU from say, a fast PCPU to a slower one or in fact, to a network core (e.g., provided by platforms with built-in network processing support like the Tolapai [40] system). This would result in freeing the faster core for use by another application. Finally, the OS is also the right locus for monitoring performance counters useful for profiling application behaviors, as shown in Figure 33.

As with operating systems, runtimes for certain application classes, such as those developed for accelerator-based applications, can provide further insights into application

behavior [19, 45]. Additional OS points can be used to convey such information to the underlying guest OS and then, the hypervisor. Examples include memory requirement based on expected inputs, or more substantive ones, such as the data needed to make decisions about whether a certain task should be run on a GPU vs. a CPU, for compute intensive vs. easily parallelizable tasks, for instance. This is important because it is the hypervisor that makes the ultimate decision about how to map VCPUs to PCPUs.

**Platform Points**: virtualization solutions like Xen [14] or Hyper-V [108] use management or privileged domains, as shown in Figure 34. They are tasked with hosting I/O backends for devices that do not support pass-through [46], accelerator backends [30], support SLAs and/or power management states [54, 95], etc., and they are responsible for providing fair shares of such resources to guests and sometimes for monitoring whether guests' SLA demands are being met. We use platform points, therefore, to implement APIs for the hardware-hypervisor interface, the hypervisor-management domain, and within the management domain, as shown in Figure 33. With these APIs, the Goal Set may include variables like the number of VCPU migrations, limits on SLA violations, or I/O fairness. Actions taken by the VMM upon violations of Goal Set parameters may convey this information to administrators, and they may cause the privileged domain to increase VCPU credits or other resources for the VM experiencing violations, or use management channels [55] to exchange such information to higher level controllers that have a more global view of how platform resources are used which would form the Data Set at this point. Another role of platform points, of course, is their use by hypervisors to convey certain hardware information to guests and vice versa, trigger hardware actions in response to guest requests.

## *6.2 Proposed System Architecture*

Performance points give rise to 'open' architectures for system software. To illustrate, consider the architectural constraints existing for the heterogeneous platforms investigated in our work, with an example shown in Figure 34:

- Accelerators may be connected in both discrete and integrated forms; there may be hybrid system configurations in which a low-power, comparatively lower performance GPU [97] is integrated with the chipset, and in addition, a higher-performance GPU [78, 41] is attached as a discrete device; and a platform may have multiple discrete GPU cards.

- There may be different operating systems driving the scalar vs. throughput-oriented cores, as exemplified by the specialized operating systems used for Intel's MIC-based processors or IBM Cell processors [96, 33]. This also means that the system environments (loaders, linkers, etc.) may differ across cores.

- Scalar and throughput-oriented cores may have different instruction set architectures (ISAs), and hence, the same code cannot be run on both kinds of cores [97].

- Even among scalar cores, as core counts increase, caching is moving from completely coherent to partially coherent to incoherent [52]. Indeed, coherence might eventually move completely to software management.

Platform configurations determine the parameters used to characterize them. These include the bandwidth and latency between different types of processors, cache coherency support, the range of supported heterogeneity, and for future, larger scale systems [16], those that characterize power consumption and software/hardware reliability.

Open hardware platforms with the above characteristics permit a wide range of applications to efficiently use their resources. Consider the multi-phase application shown in Figure 34. The figure not only depicts how the application can vary in numbers of threads, but also shows its expectations concerning its use of different kinds of processing. The performance points depicted in the figure make it possible to better meet these expectations. The points shown are similar to those in Figure 33, where PP1 is an App Point, PP2 an OS Point, and PP3 a Platform Point, as defined in Section6.1. Using these performance points, different application phases can be mapped to different hardware configurations, with the

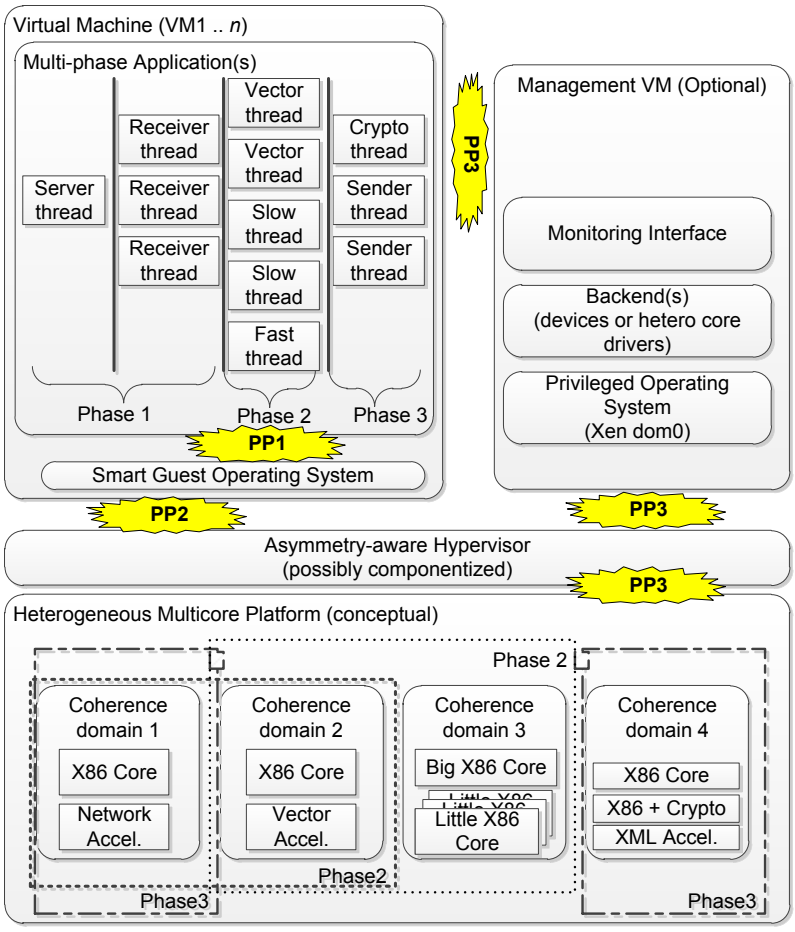



**Figure 34:** Heterogeneous Many-core System with Performance Points

help of an asymmetry-aware hypervisor and guest operating system. The asymmetry-aware hypervisor can distinguish between different guest VMs and their needs, and the guest OS can further improve performance by scheduling its threads in ways that better utilize the underlying asymmetric virtual platform. Our research has developed the asymmetry-aware hypervisor shown in the figure. The hypervisor uses a *kinship-based scheduler* to compute VCPU-PCPU mappings for a wider range of asymmetries than those reported in [44], the design and implementation details of which have already been discussed in the previous chapter. In the future, we envision that the functionality shown in Figure 34 will be componentized to accommodate increased scales, possibly disjoint ISA cores, and decreasing

coherence in the underlying platforms [75, 9, 30, 103, 52].

## *6.3 Performance Point Interface*

There are alternatives in how the conduit element of performance points can be implemented for e.g. as hypercalls, via event channels, or others. In this section, we describe some API functions that can be implemented over any of the physical channels to conduct relevant information.

**Relative change in performance** – as shown in [103], changes in scheduling credits assigned to VMs can be triggered by application-level information, such as differences in the performance experienced by an application. This change in performance could be a result of remapping of a VCPU to a different PCPU or change in phase of the application or both. Application-level information may be collected by some VM-resident runtime and conveyed to the VMM for tuning the underlying scheduler(s). For example, negative, zero or positive changes in performance can indicate whether to decrease, leave untouched, or increase the bonding between a VCPU and a PCPU over some time period. Conveying this information across to the privileged domain using a Platform Point could also trigger a request for increasing CPU or memory allocations. A function call implementing such functionality appears as:

```
convey_app_perf<Domain d, VCPU v, Time T, performance delta>
```

**Categories of execution** – can be defined for both VCPUs, depending on the expected workload, and for PCPUs, depending on the instructions supported. For example, a VCPU can be marked as VECTOR if it is going to run a vector application and extensively use Intel's SSE extensions. This can help identify VCPUs that can only be scheduled on certain processing units, as is the case for current GPGPU systems, where many contemporary GPUs do not share ISA or memory with the x86 CPUs.

```
set_vcpu_category<Domain d, VCPU v, Category c>
```

**VCPU expectations** – it is difficult for a scheduler, especially in the hypervisor, to determine when the workloads running on a particular VCPU (CPU) change phases, due to either multi-stage processing variations in input data, or changes in the kinds of algorithm being used. VCPU 'expectations' can be used to set computational expectations on a platform. For example, they may specify whether the workload to be run on a VCPU would be say, extremely cache-intensive or cpu-intensive, and typically, they will state anticipated workload behaviors, e.g., MOSTLY_CPU_INTENSIVE or MOSTLY_CACHE_INTENSIVE, rather than specify precise details such as 'I need 70% of a Xeon$^{TM}$model *xyzw* CPU clocked at 2GHz'. Their presence can be invaluable in helping the scheduler determine phase changes in workload behavior, and there are substantial applications (e.g., HPC codes) for which such hints are easily stated.

```
set_vcpu_expectation<Domain d, Vcpu v, Expectation e>
```

**Coordination requirements** – as shown in [30, 103] in the context of heterogeneous systems, coordinated scheduling is beneficial for a large class of applications, including for massively parallel workloads that synchronize with barriers or for host-accelerator applications where the host and accelerator components depend on each other for exchanging data or triggering execution. The following function can express such information via OS Points:

```
set_vcpu_buddies<Domain d, VCPU v, VCPUList vList, Integer necessity>
```

The *necessity* argument in this function call indicates the importance of a coordinated or gang scheduling effort, e.g., a high necessity value would force gang scheduling vs. a lower one causing an attempt at best effort coordination in which *v* and all VCPUs in *vList* might not be run at the same time.

**Virtual machine topology** – with kinship-based scheduling and the presence of an asymmetry-aware guest, it will become possible to change the virtual platform allocated to a guest as and when a particular set of resources becomes available, as depicted in Figure 34. The hypervisor could communicate availability to the guest using:

```
indicate_virtual_platform<Domain d, Platform p>
```

In this case, the platform *P* is a combination of various processing units, their specialities, cache and memory configurations, and interconnect information e.g. it could describe the resources shown in coherence domains 1 and 4 in Figure 34 when a domain, say VM1, needs them for Phase 3 of the application.

**Memory/cache topology** – NUMA memory properties require operating systems to increase locality of access to memory and to consider differences in access latencies when using remote memory. This has led to hypervisor-level strategies for dealing with NUMA issues [86] like CONFINE that places unaware guest VMs onto a single memory node, SPLIT for NUMA-aware guests, etc. We propose to represent such functions as performance point API calls, both due to their proven utility in improving memory utilization and performance for guests with high memory requirements and because the importance of dealing with NUMA issues will likely increase for future many-core architecture [52]. Similarly, concerning caches, past work on cache utilization suggests the existence of symbiotic vs. competing threads. The former improve overall performance by caching data for common use, whereas the latter tend to destroy each others' working sets. The following call can be used by the OS monitors when its observation of performance counters detects such thread properties:

```
set_vcpu_pair_bias<Domain d, VCPU v1, VCPU v2, Bias b>
```

As an example use, cache misses experienced by a VCPU could be due to competing threads on VCPUs or due to large data size which does not fit in cache. Depending on the situation the information sent across OS Point could be based on the function above or use the performance counters to indicate cache intensiveness of the workload(s).

**I/O Device Usage** – the authors in [46, 85] describe how fairness in scheduling I/O resources requires adjustments to existing device frontends and backends, as well as to the VCPU scheduler. Adjustments require information exchanges between guest and hypervisor that may be done with the following functions implemented by OS points:

```
read_fe_characteristics<Domain d, Frontend fe, Property p>
set_be_characteristics<Domain d, Backend be, Property p>
```

Information exchanges like these can be used to change the guest credit/priority assignments used by VMMs.

**Platform utilization information** – operating systems require hypervisor-provided information about PCPUs like performance counters, and hypervisors can make use of OS-provided information about desired hardware settings. Platform points can be used for purposes like these, with concrete examples concerning power management appearing in [73].

We have presented the evaluation of what we can achieve with some hints in Chapter 5. The kinship scheduling method utilizes hints like VCPU expectations and category to compute the VCPU-PCPU mappings.

## *6.4 Related Work*

As mentioned earlier, there is ample evidence for the utility of symbiotic execution. Cooperative interactions between guest OSes and hypervisors can cope with NUMA memory properties [86], with system power consumption [73], and can improve how memory is allocated [59] and used. Performance points generalize upon the specific APIs created in such work. Additional related work appearing in [103, 85] concerns coordinating the actions of schedulers in order to improve I/O performance, again using APIs like those offered by performance points.

[75] and [9] talk about heterogeneous systems and refer to building interfaces that implicitly (through satellite kernels) or explicitly (through messaging) help manage resources, but these do not address on-chip asymmetry, virtualization and hence, explicit application/guest participation in improving system performance.

## 6.5 Conclusions and Future Work

This chapter has introduced the concept of *Performance Points* to define a new end-to-end argument for the efficient use of heterogeneous or asymmetric multicore platforms. More explicitly, we observe that it will be necessary for future systems to reduce opaqueness between different layers of a systems software stack or possibly even implement some redundant functionality using information available at different layers, in order to achieve their performance or power goals and to scale to larger numbers of possibly heterogeneous and more distributed resources. We formalize the notion of performance points idea and present a consolidated and extended set of APIs that can provide the conduits for the information and control transfers associated with performance points. We also characterize the information that could be used to achieve the goals in a performance point. As demonstrated with related work and via experimental evaluations of asymmetric many core platforms, performance points and their use result in improved performance through higher degrees of coordination between the different layers of the systems software stack.

**Thesis discussion:** Performance points tie to the last part of the thesis statement which states that, additional benefits are derived from runtime sharing of information about hardware state and application phases across different levels of the software stack. This has not only been proven by Montage evaluation, but has enough empirical support data from other related efforts, justifying a more formalized approach.

**Future work:** A complete implementation of performance points will be evaluated with a wider variety of applications on heterogeneous hardware in future work. The set of APIs discussed here could also be extended using history information gathered over multiple runs of say, a VM that is always launched with the same application or runs of applications monitored by a VM over a certain time period in it's lifetime. Static program analysis techniques as well as runtime dynamic analysis of applications could provide good performance points to be conveyed to the guest OS in a VM which could further transmit

this data to the hypervisor for efficient scheduling. This would really enhance the knowledge accounted for by the kinship equations described in Chapter 5.

# CHAPTER VII

# PREVIOUS WORK

The importance of dealing with the heterogeneity of future multi-core platforms is widely recognized. The previous chapters have covered related work pertinent to the system being described. There are other ongoing efforts in the systems community that are currently exploring the restructuring of systems software to keep up with the evolution of hardware innovations. This dissertation has talked about one such effort which makes our approach unique in the concepts proposed and evaluated. In this chapter, we refer to efforts that have some overlap with our research goals. However, to the best of our knowledge, there are no other published efforts that have looked at developing infrastructure that can target the broad spectrum of asymmetric and heterogeneous hardware.

Cypress [24] has expressed the design principles for hypervisors actually realized in Pegasus (e.g., partitioning, localization, and customization), but Pegasus also articulates and evaluates the notion of coordinated scheduling and in addition, Pegasus does not change guest OSs, beyond introducing a loadable kernel module. Multikernel [9] and Helios [75] change system structures for multicores, advocating distributed system models and satellite kernels for processor groups, respectively. In comparison, Pegasus retains the existing operating system stack, then uses virtualization to adapt to diverse underlying hardware, and finally, leverages the federation approach shown scalable in other contexts to deal with multiple resource domains. Montage addresses the scheduling for asymmetries on-chip which pose a greater challenge especially in the presence of shared-ISA hardware. Neither Multikernel nor Helios address such platforms. Other projects in our group, namely Ocelot [17] and Shadowfax [70] can be combined to assist personality scheduling or build on top of the existing infrastructure.

# CHAPTER VIII

# CONCLUSION

The HyVM architecture supports efficient execution of Hybrid Virtual Machines on asymmetric and heterogeneous hardware. Its novel concept of VM personalities explicitly describes a HyVM's abilities to run across processors with overlapping, i.e., slightly different, and with unique ISAs specialized for certain programming models and hardware. Its multiple implementation methods cope with both shared-ISA vs. different ISA architectures, and they can run VMs with single, multiple, or with dynamically changing personalities, all without requiring changes to operating systems or applications. The performance benefits demonstrated on accelerator-based systems through Pegasus as well as more predictable and dynamic scheduling evaluated in Montage justifies our proposition for building future systems software with personality scheduling methods. Through the results discussed in this dissertation, we can re-assert the thesis statement made at the beginning that, for efficient utilization of future heterogeneous platforms:

- their ISA and architectural differences should be recognized at each level of the systems software stack;
- system mechanisms and abstractions should support explicit management of heterogeneous resources to meet application needs and platform requirements;
- heterogeneity-aware runtime methods for managing systems' resources can substantially improve resource utilization and application performance, including those that coordinate resource management across different system silos, when managing architectures in which accelerators and general purpose processors reside on different chips or in different coherence domains; and

- additional benefits are derived from runtime sharing across different levels of the software stack about hardware and state and application or workload information.

Our current Pegasus and Montage scheduler implementations are platform-specific. However, the notion of personalities and personality scheduling provides a common framework for implementing future schedulers able to operate both, on heterogeneous as well as asymmetric platforms. In this framework, there may be static personalities and kinship associations, such as with GPUs, and dynamic personalities that change over time as with the other performance and functional asymmetries addressed by Montage. We posit that the concept of personalities even extends to new architectures, like IBM's Wirespeed Processor [43], which offer hardware-level support for what HyVM's hypervisor implements in software: the hardware itself supports a VM's dynamic change in personality by automatically mapping certain 'slow' or 'long' instructions to specific accelerators. The hypervisor scheduling can then learn about such a VCPU that waits for long instructions to complete, mark its personality as REMOTE_ACCELERATED and schedule some other suitable VCPU on the corresponding PCPU to better use its idle cycles while waiting for the 'long' instruction to return. Due to closed interfaces to such acceleration units, we currently do not have the option of running the same scheduling logic on the accelerator as well, but we can surely improve the utilization of the general purpose core.

While HyVM hypervisor-level software can be viewed as emulating part of the functionality that is only present in the most recent heterogeneous hardware [43], its key contribution is that it provides a testing ground and experimental vehicle for understanding and evaluating (1) ways to make hardware components more easily managed and thus, more flexibly useful for applications, and (2) new methods for managing the resources on future heterogeneous hardware. Concerning (1), there are opportunities to experiment with fault-and-migrate vs. emulation methods for shared-ISA cores, and with emulation methods for running accelerator tasks on general purpose cores. In addition, JIT binary translation can be used to map tasks across multiple generations of accelerators. Concerning (2), we

have experimented with resource management techniques that *coordinate* across the multiple management domains present for accelerator vs. general purpose cores, and we have developed additional techniques for scheduling HyVMs on performance plus functionally asymmetric cores.

# CHAPTER IX

# FUTURE WORK

Despite the wide array and far-reaching solutions evaluated in this dissertation, there remain quite a few interesting problems to be addressed, including how to efficiently cope with runtime change in HyVM behavior, as present for VMs with multiple execution phases. Also, we have currently used performance points only with Montage to provide the kinship scheduler with hints regarding the workloads executing on VCPUs. There is therefore a question of what to put in practice and experiment, with the 'performance points' approach suggested in Chapter 6, (1) to enable applications and guests to more efficiently exploit heterogeneous hardware not just with respect to computational resources but with regard to cache, memory and interconnect heterogeneties as well, (2) permit personality scheduling methods to interact better with the resource managers controlling different processor types, (3) provide methods to incorporate service level agreements that form an important specification in popular and relevant environments like clouds and data-centers, and (4) evaluate these solutions at larger scale with power constraints since future platforms will provide large number of cores on a die with the expectation of supporting larger number of computations per unit time spent and power consumed. We have discussed some future work in individual chapters for the corresponding systems already. In this chapter, we provide a formalization for a personality scheduler that can combine lessons learnt from Pegasus and Montage to create a systems resource manager suitable for future hybrid systems.

## *9.1 Personality Scheduler*

The following functional properties can guide the development of successful and efficient HyVM personality scheduler for future systems:

$isTrans\left(V_k^{P'\rightarrow P}\right)$: Since we can morph VM personalities to run on different targets, its necessary to figure out its feasibility and benefits. Let $TT\left(V_k^{P\rightarrow P'}\right)$ be the translation overhead. Now, if $WT\left(V_k^P\right)$ is the wait time for a particular $V_k^P$ to run on some $C_j^T$, then $isTrans\left(V_k^{P\rightarrow P'}\right)$ is $True$ **if** $T \equiv T' \| \{ET\left(V_k^P\right) + WT\left(V_k^P\right) \leq ET\left(V_{k'}^{P'}\right) + WT\left(V_{k'}^{P'}\right) + TT\left(V_k^{P\rightarrow P'}\right)\}$ and $False$ otherwise, where $isTrans\left(V_k^{P\rightarrow P'}\right)$ indicates whether it is possible and reasonable to map $V_k^P$ from $C_j^T$ to some $C_{j'}^{T'}$.

$Schedule_j$: Let $Alloc_j^T[t] = V_k^P | P_j$ such that $Alloc_j^T[t] = V_k^P$ for some $V_k^P \in D_i$, and $t$ is some time interval in which a certain VCPU is mapped to some PCPU $C_j$. Since all the processing units in the platform, including accelerators, are treated as first class citizens, we can assign a $ReadyQ_j$ which is a queue of $V_k^P \in C_j^T$, i.e., schedulable units, and a function $Schedule_j$ to each processing unit in the system. Let $ET\left(V_k^P\right)$ be the estimated execution time for some $V_k^P$ on some CPU $C_j | Tag\left(C_j\right) = T$. This estimate can be based on (1) hints provided by the user, (2) history information maintained by the schedulers, or (3) runtime monitoring.

With this formulation, the assignment of all $V_k^P$ to some $ReadyQ_j$ during some accounting period $p$ is determined by the load on $C_j$, $ET\left(V_k^P\right)$ on $C_j$ and $isTrans\left(V_k^{P'\rightarrow P}\right)$, if $T'$ was the initial preferable personality which could not be activated for some reason. Since we have worked with credit-based schedulers, due in part to our use of Xen, $ReadyQ_j$ is ordered based on credits of $V_k$ assigned to $ReadyQ_j$ at any time $t$. $Schedule_j$ then simply picks $Head\left(ReadyQ_j\right)$ for execution at any time $t$.

$Coordinate\left\{C^T, C^{T'}\right\}$: For coordinating two personality scheduling domains, the $Schedule_j$ in say personality scheduling domain tagged $T$ is modified to pick a $V_k^P \in C^T$ for $D_i$ based on other $V_{k'} \in C^{T'}$ for $D_i$ such that $V_{k'}$ has been chosen to run on core tagged $T'$.

By implementing these abstractions on a platform that combines both the kinds of systems explored in this dissertation and as shown in Chapter 3-Figure 3, we can have a solution that works efficiently with good utilization on evolving hardware, requires minimal changes to applications and combines knowledge at different levels of systems stack

to perform dynamic adaptations in tandem with changing application phases. However, achieving this vision will require a concerted effort from the various silos such as developers of programming languages, various runtimes, operating systems, hypervisor and most importantly, the hardware vendors.

# REFERENCES


[1] "10 gigabit ethernet." http://en.wikipedia.org/wiki/10_Gigabit_Ethernet.

[2] "Parallel @ illinois." http://www.parallel.illinois.edu/cloud.html.

[3] AMAZON INC., "High Performance Computing Using Amazon EC2." http://aws.amazon.com/ec2/hpc-applications/.

[4] AMD, INC., "The AMD Fusion Family of APUs." http://sites.amd.com/us/fusion/apu/Pages/fusion.aspx, 2011.

[5] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., and VASUDEVAN, V., "Fawn: a fast array of wimpy nodes," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, (Big Sky, USA), October 2009.

[6] APACHE, "What Is Apache Hadoop?." http://hadoop.apache.org.

[7] BAKHODA, A., YUAN, G. L., FUNG, W. W. L., WONG, H., and AAMODT, T. M., "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *International Symposium on Performance Analysis of Systems and Software*, (Boston, USA), April 2009.

[8] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., and WARFIELD, A., "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, (Bolton Landing, USA), October 2003.

[9] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., and SINGHANIA, A., "The multikernel: a new OS architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, (Big Sky, USA), October 2009.

[10] BEAVER, D., KUMAR, S., LI, H. C., SOBEL, J., and VAJGEL, P., "Finding a needle in haystack: facebook's photo storage," in *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation*, (Vancouver, BC, Canada), October 2010.

[11] BERGMANN, A., "The Cell Processor Programming Model," in *LinuxTag*, 2005.

[12] BIENIA, C. and LI, K., "Parsec 2.0: A new benchmark suite for chip-multiprocessors," in *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[13] BORDAWEKAR, R., BONDHUGULA, U., and RAO, R., "Believe It or Not! Multi-core CPUs Can Match GPU Performance for FLOP-intensive Application!," Tech. Report RC24982, IBM T. J. Watson Research Center,, 2010.

[14] CHISNALL, D., *The Definitive Guide to the Xen Hypervisor*. Prentice Hall, 1st ed., 2008.

[15] CLARK, D. D., "The structuring of systems using upcalls," in *Proceedings of the tenth ACM Symposium on Operating Systems Principles*, (Orcas Island, United States), December 1985.

[16] DARPA, "Ubiquitous high performance computing." `https://www.fbo.gov/utils/view?id=914fa5f0a69d7bedce157d916cc97b6e`, 2010.

[17] DIAMOS, G., KERR, A., YALAMANCHILI, S., and CLARK, N., "Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems," in *Proceedings of the 19th international conference on Parallel Architectures and Compilation Techniques*, (Vienna, Austria), September 2010.

[18] DIAMOS, G. and YALAMANCHILI, S., "Speculative execution on Multi-GPU systems," in *International Parallel and Distributed Processing Symposium/International Parallel Processing Symposium*, (Atlanta, USA), April 2010.

[19] DIAMOS, G. and YALAMANCHILI, S., "Harmony: An Execution Model and Runtime for Heterogeneous Many Core Systems," in *The International ACM Symposium on High-Performance Parallel and Distributed Computing - Hot Topics*, (Boston, USA), June 2008.

[20] DOKKEN, T., HAGEN, T. R., and HJELMERVIK, J. M., "The gpu as a high performance computational resource," in *SCCG: Spring Conference on Computer graphics*, (Budmerice, Slovakia), 2005.

[21] DOWTY, M. and SUGERMAN, J., "GPU Virtualization on VMware's Hosted I/O Architecture," in *First Workshop on I/O Virtualization*, (San Diego, USA), December 2008.

[22] DOYLE, J. C., FRANCIS, B. A., and TANNENBAUM, A. R., *Feedback Control Theory*. Dover Publications, 2009.

[23] DU, P., LUSZCZEK, P., and DONGARRA, J., "OpenCL Evaluation for Numerical Linear Algebra Library Development," in *Symposium on Application Accelerators in High Performance Computing*, (Knoxville, USA), July 2010.

[24] FEDOROVA, A., KUMAR, V., KAZEMPOUR, V., RAY, S., and ALAGHEB, P., "Cypress: A Scheduling Infrastructure for a Many-Core Hypervisor," in *In Proceedings of the 1st Workshop on Managed Many-Core Systems*, (Boston, USA), June 2008.

[25] FRASER, K., H, S., NEUGEBAUER, R., PRATT, I., WARFIELD, A., and WILLIAMSON, M., "Safe hardware access with the xen virtual machine monitor," in *In 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure*, (Boston, MA), October 2004.

[26] GAMSA, B., KRIEGER, O., APPAVOO, J., and STUMM, M., "Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System," in *In Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, (New Orleans, USA), February 1999.

[27] GOVIL, K., TEODOSIU, D., HUANG, Y., and ROSENBLUM, M., "Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors," in *Proceedings of the seventeenth ACM Symposium on Operating Systems Principles*, (Charleston, USA), December 1999.

[28] GUERON, S., "Intel advanced encryption standard (aes) instructions set - rev 3." `http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set/`, 2010.

[29] GUEVARA, M., GREGG, C., HAZELWOOD, K., and SKADRON, K., "Enabling Task Parallelism in the CUDA Scheduler," in *Programming Models for Emerging Architectures*, (Raleigh, USA), September 2009.

[30] GUPTA, V., GAVRILOVSKA, A., SCHWAN, K., KHARCHE, H., TOLIA, N., TALWAR, V., and RANGANATHAN, P., "GViM: GPU-accelerated Virtual Machines," in *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, (Nuremberg, Germany), March 2009.

[31] GUPTA, V., KNAUERHASE, R., and SCHWAN, K., "Attaining System Performance Points: Revisiting the End-to-End Argument in System Design for Heterogeneous Many-core Systems," *SIGOPS Operating Systems Review*, vol. 45, January 2011.

[32] GUPTA, V., SCHWAN, K., TOLIA, N., and OTHERS, "Pegasus: Coordinated scheduling for virtualized accelerator-based systems," in *USENIX ATC*, (Portland, USA), 2011.

[33] GUPTA, V., XENIDIS, J., TEMBEY, P., SCHWAN, K., and GAVRILOVSKA, A., "Cellule: Lightweight Execution Environment for Accelerator-based Systems," Tech. Rep. GIT-CERCS-10-03, Georgia Tech, March 2010.

[34] GUPTA, V. and NATHUJI, R., "Analyzing Performance Asymmetric Multicore Processors for Latency Sensitive Datacenter Applications," in *Proceedings of the 2010 international conference on Power Aware Computing and Systems*, (Vancouver BC, Canada), October 2010.

[35] HEINIG, A., STRUNK, J., REHM, W., and SCHICK, H., *ACCFS - Operating System Integration of Computational Accelerators Using a VFS Approach*, vol. 5453. Springer Berlin, 2009.

[36] HENNING, J. L., "Spec cpu2006 benchmark descriptions," *SIGARCH Computer Architecture News*, vol. 34, pp. 1–17, September 2006.

[37] HOURD, J., FAN, C., ZENG, J., ZHANG, Q. S., BEST, M. J., FEDOROVA, A., and MUSTARD, C., "Exploring Practical Benefits of Asymmetric Multicore Processors," in *The 2nd Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures*, (Austin, USA), June 2009.

[38] IBM INC., "Accelerated Library Framework for Cell Broadband Engine Programmer's Guide and API Reference." `http://moss.csc.ncsu.edu/~mueller/cluster/ps3/SDK3.0/docs/lib/ALF_Prog_Guide_API_v3.0.pdf`, October 2007.

[39] INTEL CORP., "Intel AESNI Sample Library." `http://software.intel.com/en-us/articles/download-the-intel-aesni-sample-library/`, 2010.

[40] INTEL CORPORATION, "Enabling Consistent Platform-level Services for Tightly Coupled Accelerators." `http://download.intel.com/technology/platforms/quickassist/quickassist_aal_whitepaper.pdf`.

[41] INTEL NEWS RELEASE, "Intel Unveils New Product Plans for High-Performance Computing." `http://www.intel.com/pressroom/archive/releases/2010/20100531comp.htm`.

[42] JIMÉNEZ, V. J., VILANOVA, L., GELADO, I., GIL, M., FURSIN, G., and NAVARRO, N., "Predictive Runtime Code Scheduling for Heterogeneous Architectures," in *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, (Paphos, Cyprus), January 2009.

[43] JOHNSON, C., ALLEN, D. H., BROWN, J., VANDERWIEL, S., HOOVER, R., ACHILLES, H., CHER, C.-Y., MAY, G. A., FRANKE, H., XENIDIS, J., and BASSO, C., "A Wire-Speed PowerTM Processor: 2.3GHz 45nm SOI with 16 Cores and 64 Threads," in *IEEE International Solid-State Circuits Conference*, (San Francisco, USA), February 2010.

[44] KAZEMPOUR, V., KAMALI, A., and FEDOROVA, A., "AASH: an asymmetry-aware scheduler for hypervisors," in *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, (Pittsburgh, USA), March 2010.

[45] KERR, A., DIAMOS, G., and YALAMANCHILI, S., "A Characterization and Analysis of PTX Kernels," in *IEEE International Symposium on Workload Characterization*, (Austin, USA), October 2009.

[46] KESAVAN, M., GAVRILOVSKA, A., and SCHWAN, K., "Differential Virtual Time (DVT): Rethinking I/O Service Differentiation for Virtual Machines," in *Proceedings of the 1st ACM symposium on Cloud computing*, (Indianapolis, USA), June 2010.

[47] KHRONOS GROUP, "The OpenCL Specification." `http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf`, Dec. 2008.

[48] KNAUERHASE, R., BRETT, P., HOHLT, B., LI, T., and HAHN, S., "Using OS Observations to Improve Performance in Multicore Systems," *IEEE Micro*, vol. 28, no. 3, 2008.

[49] KOH, Y., KNAUERHASE, R. C., BRETT, P., BOWMAN, M., WEN, Z., and PU, C., "An Analysis of Performance Interference Effects in Virtual Environments," in *Proceedings of International Symposium on Performance Analysis of Systems and Software*, April 2007.

[50] KOUFATY, D., REDDY, D., and HAHN, S., "Bias scheduling in heterogeneous multi-core architectures," in *Proceedings of the 5th European conference on Computer systems*, (Paris, France), April 2010.

[51] KRASIC, C., SAUBHASIK, M., SINHA, A., and GOEL, A., "Fair and timely scheduling via cooperative polling," in *Proceedings of the 4th ACM European conference on Computer systems*, (Nuremberg, Germany), April 2009.

[52] KUBASKA, T., "SCC Platform Overview." `http://communities.intel.com/docs/DOC-5512`.

[53] KUMAR, R., TULLSEN, D. M., RANGANATHAN, P., JOUPPI, N. P., and FARKAS, K. I., "Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance," in *Proceedings of the 31st annual international symposium on Computer Architecture*, (München, Germany), June 2004.

[54] KUMAR, S., TALWAR, V., KUMAR, V., RANGANATHAN, P., and SCHWAN, K., "vManage: Loosely Coupled Platform and Virtualization Management in Data Centers," in *The 6th International Conference on Autonomic Computing and Communications*, (Barcelona, Spain), June 2009.

[55] KUMAR, S., TALWAR, V., RANGANATHAN, P., NATHUJI, R., and SCHWAN, K., "M-Channels and M-Brokers: Coordinated Management in Virtualized Systems," in *Proceedings of the Workshop on Managed Many-Core Systems*, 2008.

[56] KWON, Y., KIM, C., MAENG, S., and HUH, J., "Virtualizing performance asymmetric multi-core systems," in *Proceeding of the 38th annual international symposium on Computer Architecture*, (San Jose, USA), June 2011.

[57] LAGAR-CAVILLA, H. A., TOLIA, N., SATYANARAYANAN, M., and DE LARA, E., "VMM-independent graphics acceleration," in *Proceedings of the 3rd international conference on Virtual Execution Environments*, (San Diego, USA), June 2007.

[58] LAKSHMINARAYANA, N., RAO, S., and KIM, H., "Asymmetry Aware Scheduling Algorithms for Asymmetric Multiprocessors," June 2008.

[59] LANGE, J. R. and DINDA, P., "Symcall: symbiotic virtualization through vmm-to-guest upcalls," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, (Newport Beach, USA), March 2011.

[60] LEUNG, F., NEIGER, G., RODGERS, D., SANTONI, A., and UHLIG, R., "Intel Virtualization Technology: Hardware support for efficient processor virtualization," *Intel Technology Journal*, vol. 10, August 2006.

[61] LEVON, J., "Oprofile manual." `http://oprofile.sourceforge.net/doc/index.html`, 2000.

[62] LI, T., BAUMBERGER, D., and HAHN, S., "Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, (Raleigh, USA), February 2009.

[63] LI, T., BAUMBERGER, D., KOUFATY, D. A., and HAHN, S., "Efficient operating system scheduling for performance-asymmetric multi-core architectures," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, (Reno, USA), November 2007.

[64] LI, T., BRETT, P., HOHLT, B., KNAUERHASE, R., MCELDERRY, S. D., and HAHN, S., "Operating system support for shared-isa asymmetric multi-core architectures," in *Proceedings of the Fourth Annual Workshop on the Interaction between Operating Systems and Computer Architecture*, (Beijing, China), June 2008.

[65] LI, T., BRETT, P., KNAUERHASE, R., KOUFATY, D. A., REDDY, D., and HAHN, S., "Operating system support for overlapping-ISA heterogeneous multi-core architectures," in *Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture*, (Bangalore, India), January 2010.

[66] LI, Z., BAI, Y., ZHANG, H., and MA, Y., "Affinity-Aware Dynamic Pinning Scheduling for Virtual Machines," in *Proceedings of 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, (Indianapolis, USA), December 2010.

[67] LINDERMAN, M. D., COLLINS, J. D., WANG, H., and MENG, T. H., "Merge: a programming model for heterogeneous multi-core systems," in *Proceedings of the 13th international conference on Architectural Support for Programming Languages and Operating Systems*, (Seattle, USA), March 2008.

[68] LUK, C.-K., HONG, S., and KIM, H., "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, (New York, USA), December 2009.

[69] MARCIAL, E., "The ICE Financial Application." http://www.theice.com, August 2010. Private Communication.

[70] MERRITT, A. M., GUPTA, V., VERMA, A., GAVRILOVSKA, A., and SCHWAN, K., "Shadowfax: scaling in heterogeneous cluster systems via GPGPU assemblies," in *Proceedings of the 5th international workshop on Virtualization technologies in distributed computing*, (San Jose, USA), June 2011.

[71] MICROSOFT CORP., "What is Photosynth?." http://photosynth.net/about.aspx, 2010.

[72] MORAD, T. Y., WEISER, U., KOLODNY, A., VALERO, M., and AYGUAD, E., "Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors," *IEEE Computer Architecture Letters*, vol. 5, January 2006.

[73] NATHUJI, R. and SCHWAN, K., "VirtualPower: coordinated power management in virtualized enterprise systems," in *Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, (Stevenson, USA), October 2007.

[74] NETFLIX INC., "Netflix." http://en.wikipedia.org/wiki/Netflix.

[75] NIGHTINGALE, E. B., HODSON, O., MCILROY, R., HAWBLITZEL, C., and HUNT, G., "Helios: heterogeneous multiprocessing with satellite kernels," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating Systems Principles*, (Big Sky, USA), October 2009.

[76] NVIDIA CORP., "NVIDIA's Next Generation CUDA Compute Architecture: Fermi." http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[77] NVIDIA CORP., "NVIDIA CUDA Compute Unified Device Architecture." http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf, June 2007.

[78] NVIDIA CORP., "NVIDIA Tesla C870." http://www.nvidia.com/object/tesla_c870.html, Dec. 2007.

[79] ONLIVE INC., "OnLive." http://www.onlive.com.

[80] OPENCIRRUS, "Open Cirrus." http://www.opencirrus.org/.

[81] OPENFABRICS GROUP, "OpenFabrics." http://www.openfabrics.org.

[82] OW2 CONSORTIUM, "RUBiS: Rice University Bidding System." http://rubis.ow2.org/index.html.

[83] RACKSPACE INC., "Rackspace hosting." http://www.rackspace.com/.

[84] RAJ, H. and SCHWAN, K., "High performance and scalable I/O virtualization via self-virtualized devices," in *Proceedings of the 16th international symposium on High Performance Distributed Computing*, (Monterey, USA), June 2007.

[85] RANADIVE, A., GAVRILOVSKA, A., and SCHWAN, K., "FaReS: Fair Resource Scheduling for VMM-Bypass InfiniBand Devices," in *Proceedings of 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, (Melbourne, Australia), May 2010.

[86] RAO, D. and SCHWAN, K., "vNUMA-mgr : Managing VM Memory on NUMA Platforms," in *Proceedings of 2010 International Conference on High Performance Computing*, (Goa, India), December 2010.

[87] RAVINDRAN, B., JENSEN, E. D., and LI, P., "On recent advances in time/utility function real-time scheduling and resource management," in *Proceedings of Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, (Seattle, USA), May 2005.

[88] REINDERS, J., *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilley, 1 ed., July 2007.

[89] ROSU, D., SCHWAN, K., YALAMANCHILI, S., and JHA, R., "On Adaptive Resource Allocation for Complex Real-Time Applications," in *Proceedings of the 18th IEEE Real-Time Systems Symposium*, (San Francisco, USA), December 1997.

[90] RYOO, S., RODRIGUES, C. I., BAGHSORKHI, S. S., STONE, S. S., KIRK, D. B., and MEI W. HWU, W., "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, (Salt Lake City, USA), February 2008.

[91] SAEZ, J. C., PRIETO, M., FEDOROVA, A., and BLAGODUROV, S., "A comprehensive scheduler for asymmetric multicore systems," in *Proceedings of the 5th European conference on Computer Systems*, (Paris, France), April 2010.

[92] SAEZ, J. C., SHELEPOV, D., FEDOROVA, A., and PRIETO, M., "Leveraging workload diversity through os scheduling to maximize performance on single-isa heterogeneous multicore systems," *Journal of Parallel and Distributed Computing*, vol. 71, January 2011.

[93] SALTZER, J. H., REED, D. P., and CLARK, D. D., "End-to-end arguments in system design," *ACM Transactions on Computer Systems*, vol. 2, no. 4, pp. 277–288, 1984.

[94] SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G. J., and PRATT, I., "Bridging the Gap between Software and Hardware Techniques for I/O Virtualization," in *Proceedings of USENIX 2008 Annual Technical Conference*, (Boston, MA), June 2008.

[95] SANTOS, J. R., TURNER, Y., and MUDIGONDA, J., "Taming Heterogeneous NIC Capabilities for I/O Virtualization," in *Workshop on I/O Virtualization*, (San Diego, CA), December 2008.

[96] SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., and HANRAHAN, P., "Larrabee: a many-core x86 architecture for visual computing," *ACM Transactions on Graphics*, vol. 27, no. 3, 2008.

[97] SHIMPI, A. L., "Intel's Sandy Bridge Architecture Exposed." `http://www.anandtech.com/show/3922/intels-sandy-bridge-architecture-exposed`.

[98] SNAPFISH, "About Snapfish." `http://www.snapfish.com`.

[99] SNAVELY, N., SEITZ, S. M., and SZELISKI, R., "Modeling the World from Internet Photo Collections," *International Journal of Computer Vision*, vol. 80, November 2008.

[100] SREERAM, J. and PANDE, S., "GLIMPSES: A Profiling Tool for Rapid SPE Code Prototyping," in *Workshop on New Horizons in Compilers*, December 2007.

[101] STRATTON, J., STONE, S., and MEI HWU, W., "MCUDA: An Efficient Implementation of CUDA Kernels on Multi-cores," Tech. Rep. IMPACT-08-01, University of Illinois at Urbana-Champaign, March 2008.

[102] TANG, L., MARS, J., and SOFFA, M. L., "Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures," in *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, (San Jose, USA), June 2011.

[103] TEMBEY, P., GAVRILOVSKA, A., and SCHWAN, K., "A case for coordinated resource management in heterogeneous multicore platforms," in *Proceedings of 6th Workshop on the Interaction between Operating Systems and Computer Architecture*, (St. Malo, France), June 2010.

[104] TEODORO, G., HARTLEY, T. D. R., CATALYUREK, U., and FERREIRA, R., "Runtime optimizations for replicated dataflows on heterogeneous environments," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, (Chicago, Illinois), June 2010.

[105] TORRELLAS, J., TUCKER, A., and GUPTA, A., "Benefits of cache-affinity scheduling in shared-memory multiprocessors: a summary," *SIGMETRICS Performance Evaluation Review*, vol. 21, June 1993.

[106] TURNER, J. A., "The Los Alamos Roadrunner Petascale Hybrid Supercomputer: Overview of Applications, Results, and Programming." Roadrunner Technical Seminar Series, 2008.

[107] UHLIG, V., LEVASSEUR, J., SKOGLUND, E., and DANNOWSKI, U., "Towards scalable multiprocessor virtual machines," in *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*, (San Jose, California), May 2004.

[108] VELTE, A. and VELTE, T., *Microsoft Virtualization with Hyper-V*. McGraw-Hill, Inc., 2010.

[109] VETTER, J., GLASSBROOK, D., DONGARRA, J., FUJIMOTO, R., SCHULTHESS, T., and SCHWAN, K., "Keeneland - Enabling Heterogeneous Computing For The Open Science Community." `http://www.nvidia.com/content/PDF/sc_2010/theater/Vetter_SC10.pdf`, November 2010.

[110] VMWARE CORP., "VMware vSphere 4: The CPU Scheduler in VMware ESX 4." `http://www.vmware.com/files/pdf/perf-vsphere-cpu_scheduler.pdf`, 2009.

[111] VOLKOV, V. and DEMMEL, J., "LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs," Tech. Rep. UCB/EECS-2008-49, EECS Department, University of California, Berkeley, May 2008.

[112] WALDSPURGER, C. A., "Memory Resource Management in VMware ESX Server," in *Proceedings of the 5th symposium on Operating Systems Design and Implementation*, (Boston, USA), December 2002.

[113] WEISSTEIN, E., "H-spread." `http://mathworld.wolfram.com/H-Spread.html`.

[114] WILLIAM D. NORCOTT, "Iozone Filesystem Benchmark." `http://www.iozone.org`.

[115] YU, W. and VETTER, J. S., "Xen-Based HPC: A Parallel I/O Perspective," in *Proceedings of 8th IEEE International Symposium on Cluster Computing and the Grid*, (Lyon, France), May 2008.

# VITA

Vishakha Gupta was born to Dr. Suresh Gupta and Mrs. Vijaya Gupta on March 2nd, 1983 and brought up in Nagpur, a city in Maharashtra, India. She did her high school in Nagpur before moving to Pilani in Rajasthan for a Bachelor in Computer Science from BITS, Pilani, India. After BE, she completed Masters in Information Networking from Carnegie Mellon University, Pittsburgh, USA. She completed her PhD from College of Computing in Georgia Institute of Technology, Atlanta, USA working with Prof. Karsten Schwan. Vishakha is now a Research Scientist at Intel Labs in Hillsboro, OR and lives there with her husband Romain Cledat.

Her research interests encompass systems in general with particular inclination towards virtualization, embedded and real time systems as well as distributed systems. She loves to work on systems which impose stringent requirements in terms of software design and coding and call for innovative solutions. It is this love and extreme fascination for challenges that has drawn her into the amazing world of research!